Gene-Environment Interaction Analysis using GPU

Daniel Berglund

November 2014

Abstract

Abstract

Contents

Chapter 1

Introduction

1.1 Outline

The first chapter contains a short introduction to the area and the terminology. The second chapter gives the theoretical background. It starts with the statistics and algorithms and then describes the computer architecture and software design. The third algorithm section explains the structure of the current algorithms and also explains the design of the program that was developed in this thesis called CuEira. Section four provides the results and comparisons of the program performance. The fifth section is the discussion of these results. The last section discusses the future work.

1.2 Genome-wide association studies

One type of study to find associations between genetic markers and diseases or other traits in different individuals is genome-wide association study(GWAS). Most GWAS do not study interactions between the genetic markers or between the genetic markers and environmental factors [1, 2]. Investigating gene-gene interactions recently has become more common[1], however gene-environment interactions are still uncommon in studies[2]. Interactions between genes and environmental factors are considered to be important for complex diseases such as cancer or autoimmune diseases[1, 2, 3, 4]. In general a complex disease develops due to a combination of factors, not just a single gene or environmental factor[5]. Examples of environmental factors are smoking, physical activity and so on.
GWAS usually has a study design that is either cohort or case-control[5]. In cohort study a sample of a population is followed and over time some individuals will develop the disease of interest[6]. In case-control studies two groups are compared with each other to find the risk factors[6]. One of these two groups consists of individuals with the disease and the other group consists of individuals similar to the first group but who do not have the disease[6].

A typical GWAS consists of tens of thousands of individuals and up to several millions of genetic markers[1, 7]. In gene-gene interaction GWAS few studies investigate more than the second order interaction because of the high number

of possible combinations. There are some algorithms that can investigate higher order interactions however these have drawbacks[8, 9, 10].

1.3 Genetics

The genetic information is stored in *chromosomes* each consisting of a *DNA molecule*[11]. Each chromosome comes in a pair where both chromosomes are nearly identical, except for the chromosomes related to gender[11]. The DNA molecule is a double stranded helix of four nucleotide bases, *adenine*(A),*cytosine*(C),*guanine*(G) and *thymine*(T). They are always paired as A-T and G-C. *Genes* are sections of these DNA molecules. The pair of genes in the chromosomes is a *genotype*. The observable product of the genotype is called *phenotype*, e.g. eye colour, fur patters.

A position of the DNA or a gene is a *locus*[11]. A variation of the same gene or locus is an *allele*[11]. The effect of an allele on a phenotype can be either *dominant*, *recessive* or *co-dominant*. Dominant means that if the allele is present the phenotype is expressed. For recessive the allele needs to be present in both chromosomes to be expressed[11]. Co-dominant is when both alleles are expressed, when different alleles are present this usually produce some kind of intermediate state[11]. An example of co-dominance is the human blood groups[11].

Genotype	Dominant	Recessive
AA	Effect	Effect
GA	Effect	No Effect
GG	No effect	No effect

Table 1.1: *The phenotype based on the genetic model and if the allele with the effect is A*

1.3.1 Major, Minor and Risk Allele

The the allele that is least common using all individuals(both cases and controls) is the *minor allele*, the one that is most common is the *major allele*. The frequency of the minor allele is the *minor allele frequency*(MAF)[12, 3]. If the MAF is too low an analysis often will be of little value so all SNPs with MAF below a threshold are usually skipped. 5% is common value of the threshold[7, 3].

To determine if the genetic risk is present for each individual one of the alleles needs to be chosen as the *risk allele*. GEIRA and JEIRA defines the risk allele as the minor allele in cases if the MAF of cases is greater than the MAF of controls, otherwise the major allele in cases is used[13, 12]. However JEIRA and GEISA does not calculate the risk allele according to that definition, see appendix C for the details. In this thesis the risk allele will be defined as the allele that has higher frequency in cases than controls. It is similar to the definition using MAF in most cases. Appendix C has more detailed comparisons.

The presence of genetic risk based on the risk allele and the genetic model can be coded as a variable than then be can used in a statistical model. For

dominant or recessive genetic model this means that the variable is binary since the risk is either present or not[3]. See table 1.1 for how the coding is based on the model. For co-dominant model the number of risk alleles in the individual is used as the variable, 0, 1 or 2 [3].

1.4 Defining Interaction

There are several ways to define interaction. The overall goal is often to detect if *biological* interactions are present. *Biological* interaction is when different factors co-operate through a physiological or biological mechanism and cause the effect, e.g. the disease. The information about the factors can then be used to explain the mechanisms involved in causing the disease and possibly help to find cures for them. However biological interaction is not well defined and thus it is not possible to calculate it directly from data.[14, 5, 15]

That is why statistical interaction is used and it is assumed that the presence of statistical interaction implies the presence of biological interaction
Statistical interaction on the other hand is well defined. However it is scale dependent, i.e. interactions can appear and disappear based on transformations of the given data. Statistical interaction also depends on the model used. The common way to define statistical interaction is to consider the presence of a product term between the variables in the model, this is referred to as *multiplicative* interaction. For instance for a linear model

$$f(x, y) = ax + by + cxy + d \tag{1.1}$$

c is the product term that represents multiplicative interaction between variables x and y so the test for multiplicative interaction is to test if c=0 [3, 14, 5]. In some cases it is instead tested as a=b=c=0, this is testing for *association allowing for interaction*[1].

Additive interaction is another kind of statistical interaction that is broader than multiplicative interaction. It is when the total effect of the interacting factors present is larger than the sum of their separate effects. A more precise definition can be found in section 2.1.3. It implies biological interaction as defined by Rothman[5], which is sometimes called *causal interdependence* or *causal interaction*[16]. Additive interaction can find more possible interactions than multiplicative and since it is closer to biological interaction it is often chosen in epidemiological studies[14].

1.5 GEIRA, JEIRA and GEISA, the Aim of the Project

To search for interaction in data various tools can be used. *GEIRA*[3] is one such tool which uses logistic regression, additive interaction and with bootstrap to asses the quality of the models[3]. It comes in two versions, one made in R and one in SAS. To improve speedup it was later implemented in Java and called *JEIRA*[12]. It is parallel which made it significantly faster[12]. *GEISA*[13] is another tool based on JEIRA however it uses permutation testing instead of

bootstrap to asses the certainty of the models[13]. The programs structure is explained in section 3, the algorithms are explained in section 2.6. Bootstrap calculates the certainty by resampling the data[17]. Permutation tests works in a similar way but only randomizes the outcomes [18].

The aim of this project is to make the gene-environment interaction analysis faster by using graphics processors and to be able to handle larger amounts of data. The program is written in C++ and CUDA.

Chapter 2

Background

This section will go through some of the background starting with the statistical background, continuing with computer architecture and software design after that. Last is a section about the algorithms.

2.1 Statistical Background

This section will explain some of the statistical background of categorical statistics.

2.1.1 Contingency Tables

A contingency table is a matrix used to describe categorical data. Each cell contains a count of occurrences for a specific combination of variables. Table 2.1 is an example of an 2 x 2 table. From this table we can for instance see that 688 smokers got lung cancer. Contingency tables are the basis for various statistical tests to model the data.[17]

	Lung cancer	No lung cancer
Smoker	688	650
Non smoker	21	59

Table 2.1: *Contingency table describing the outcome of a study, from [17], page 42*

2.1.2 Relative Risk and Odds Ratio

From a contingency table it is possible to calculate some useful measures such as the risk of getting the outcome based on exposure. The risk for a row i in the table will be referred to as π_i. For table 2.1 the risks π_1 and π_2 is

$$\pi_1 = \frac{688}{688 + 650} = 0.51 \tag{2.1}$$

$$\pi_2 = \frac{21}{21 + 59} = 0.36 \tag{2.2}$$

To compare different risks, for instance between smokers and non smokers, the ratio of the risks is used[17]. It is called *relative* risk(RR) is defined as[17]

$$RR = \frac{\pi_1}{\pi_2} \tag{2.3}$$

So for the table 2.1 the relative risk of getting lung cancer based on exposure to smoking is

$$RR = \frac{0.52}{0.36} = 1.96 \tag{2.4}$$

This means that the risk of getting lung cancer for a smoker is almost twice as high for a non smoker in this data.
Another useful measure is *odds* and *odds ratio*(OR). The odds is[17]

$$\Omega = \frac{\pi}{1-\pi} \tag{2.5}$$

The odds are non-negative and $\Omega > 1$ when the outcome is more likely than not[17]. So for $\pi = 0.75$ the odds is $\Omega = \frac{0.75}{1-0.75} = 3$. This means that the outcome is 3 times more likely to occur than not. ORs can be used as an approximation of RR in case control studies because RR can not be estimated in that type of studies[19]. Cohort studies on the other hand can give estimates of RR[19]. OR is the ratio of the odds just as RR is the ratio of the risks[17].

$$\theta = \frac{\Omega_1}{\Omega_2} \tag{2.6}$$

In the case when the outcome is a disease or similar variables with odds ratio below one are called *protective* and when it is above one it is a *risk factor*[20].

2.1.3 Additive Interaction

Interaction was talked about in section 1.4. The more precise definition of additive interaction is the divergence from additive effects on a logarithmic scale, e.g[5].

$$OR_{both\ factors\ present} > OR_{first\ factor\ present} + OR_{second\ factor\ present} - 1 \tag{2.7}$$

A third variable is used to model the interaction for both multiplicative and additive interaction. Multiplicative interaction as explained in section 1.4 does not include the effects from the factors, i.e. the main effects, in the interaction variable. However for additive interaction the interaction variable models the whole effect including the main effects. This means that the third variable for multiplicative and additive interaction are identical, however for additive interaction the first and second variable is zero when both factors are present. The coding for additive interaction is summarized in table 2.2[12].

2.1.4 Statistic Measures for Additive Interaction

Based on the statistical model and its corresponding ORs there are some measures that can be calculated. These measures show various properties of the additive interaction. They are defined using RR however as mentioned before

Factor present	First variable	Second variable	Interaction
None	0	0	0
First	1	0	0
Second	0	1	0
Both	0	0	1

Table 2.2: *Coding of the variables for LR*

in section 2.1.2 OR can be used to approximate RR.

Relative excess risk due to interaction(RERI) is how much of the risk is due to interaction[5, 20]. It is defined as[5, 20]

$$RERI = RR_{11} - RR_{10} - RR_{01} + 1 \tag{2.8}$$

Attributable proportion due to interaction(AP) is similar to RERI however is the proportion relative to the interaction relative risk[5, 20].

$$AP = \frac{RERI}{RR_{11}} = \frac{1}{RR_{11}} - \frac{RR_{10}}{RR_{11}} - \frac{RR_{01}}{RR_{11}} + 1 \tag{2.9}$$

Synergy index(SI) is the ratio of the combined effects and the individual effects[5, 20].

$$\frac{RR_{11} - 1}{RR_{10} + RR_{01} - 2} \tag{2.10}$$

By using the definition of additive interaction it can be shown that additive interaction is present when $RERI > 0$, $AP > 0$ and $SI > 0$[20].

2.1.4.1 Recoding of Protective Effects

The measures for additive interaction RERI, AP and SI explained in the previous section are developed for risk factors, i.e. $OR > 1$, which causes problems if a factor is protective[20]. This can be solved by *recoding*, it switches the group of individuals without both factors, i.e. the *reference group*, with the group that has the lowest OR[20]. There are three possible combinations of the factors excluding the reference group shown in table 2.3. Switching a group changes the level of the factors for all in individuals in the group to the reference groups level, and switches the individuals in the reference group with the groups level.

Genetic factor	Environment factor	Recoding case
0	0	N/A
1	0	1
0	1	2
1	1	3

Table 2.3: *Cases of recoding*

2.1.5 Confounders and Covariates

Confounding is one of the central issues in design of epidemiological studies. It is when the effect of the exposure is mixed with the effect of another variable. So if we do not measure the second variable, the effect of the first would be estimated as stronger than it really is. The second variable is then a *confounder*. Several methods in epidemiology are about avoiding or adjusting for confounding. Sometimes these variables needs to be incorporated into the models. *Covariates* are possible confounders or other variables that one wants to adjust for in the model. Sometimes covariates are called *control variables*.[15, 5]

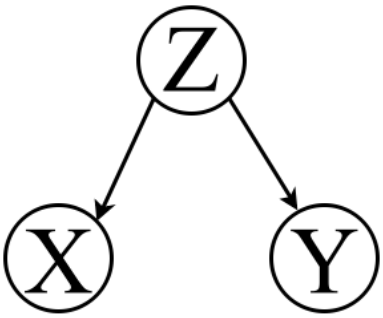


Figure 2.1: *Illustration of a simple case of confounding. If we do not observe Z we might falsely find an association between X and Y*

For instance if a study would look into the effects of yellow teeth on lung cancer they would likely find an effect. However it would be due to the confounder smoking since smoking causes yellow teeth and lung cancer[21].

2.1.6 The Multiple Testing Problem

It is not uncommon to test many hypothesis on the same data and in the case of GWAS it can be billions of tests. If one continues to test one should eventually find something that is significant due to random chance. With the common significance threshold of 5% it is expected to get 1 in 20 false positives under the assumption that the null hypothesis is true. The problem arises from the fact that the hypothesis tests are dependent on each other since they use parts of the same data. This is the multiple testing problem and if it is not corrected for many false positives might be found.[22]

Bonferroni correction is the simplest method and viewed as a conservative way to correct for this problem. It simply divides the significance threshold by the number of hypothesis tested. However the number of hypothesis made is not always clear. With a two-stage analysis, is the number of hypothesis the number of tests done in both stages combined, the number made in the first stage or the second stage?[22]

2.2 Computer Architecture Background

This section explains some of the computer architectures used to perform tasks in a thesis. A large part of this chapter is focused on the optimization techniques that are in use. For the purpose of this thesis it is best to think of the computer consisting of four main parts, the processor, the data storage, the memory and the accelerator.

2.2.1 Central Processing Unit

Central processing unit(CPU) is the part of the computer that executes instructions one by one[23]. Most modern CPUs have multi-core architecture meaning they consist of several processors[23]. Each core can perform tasks independent of each other. Multi-core architecture affects the way the programs are made because they need to be parallel to get maximum speed. More about this in section 2.2.2.
The figures 2.2 and 2.5 shows how the CPU is divided into areas and that most of it is not used for calculations. A large part of the area is used for various optimizations.

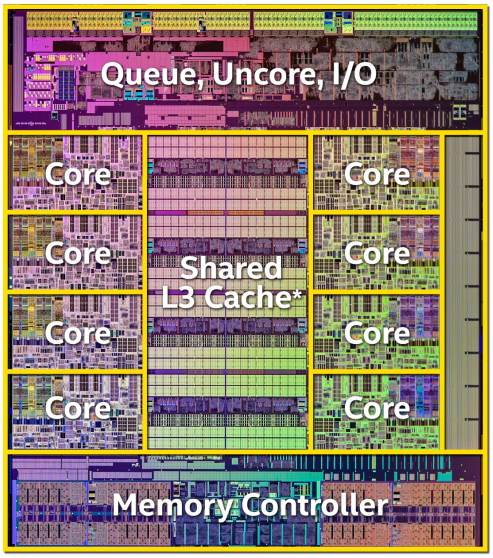


Figure 2.2: *The layout of the Haswell architecture i7-5960x, from [24]*

2.2.2 Concurrency and Threads

Concurrency means doing multiple things and the same time, instead of *sequential*. This can cause various problems such as the same object being accessed at the same time. The dinning philosophers is an concurrency problem that can used to illustrate some of these problems. A group of philosophers is sitting at a round table, each has a plate of spaghetti in front of them and there is a fork between each pair of philosophers[25]. The philosophers alternate between

thinking and eating. However they can only eat if they have both the right and left fork[25].

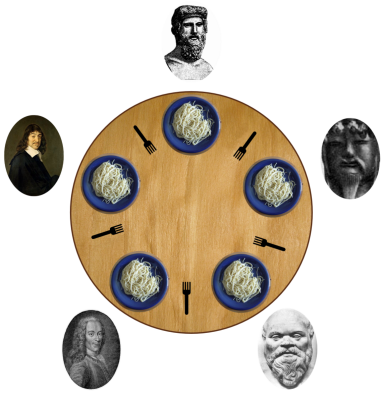


Figure 2.3: *Illustration of the dining philosophers problem. Wikipedia Commons*

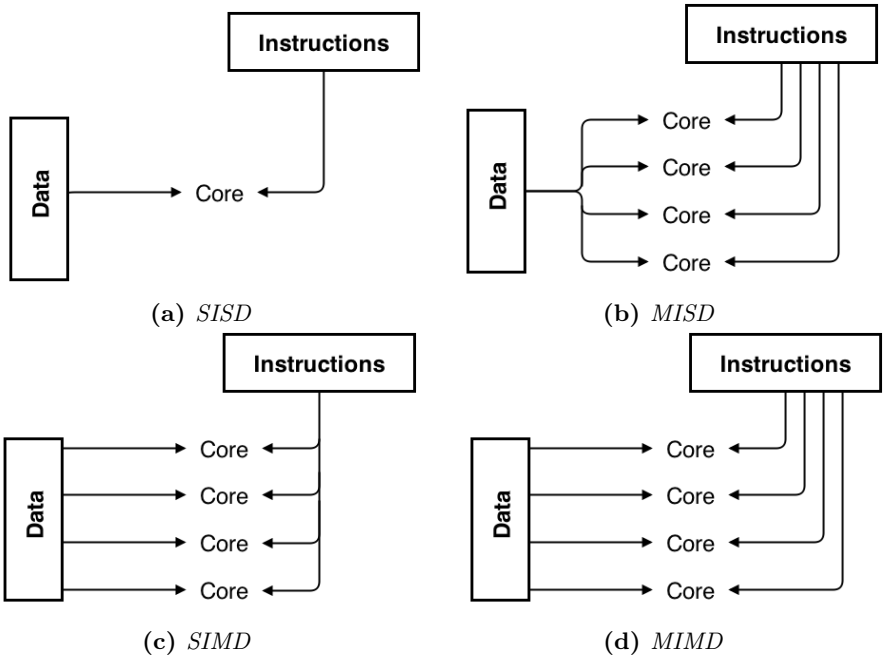
A potential proposal[25] for behaviour instructions could be:

- Think
- Wait for a fork to become available and pick up that fork
- Wait for and pick up the other fork
- Eat
- Put down the forks one by one
- Go back to thinking

The problem is that this set of instructions can lead to a state where everyone is holding one fork and waiting to get the other one[25]. But no one is done eating so no forks will become available. The system is then locked into its state, this is called a *deadlock*[25, 23]. A potential solution to the problem in this case is to introduce a person that dictates if a philosopher is allowed to eat or not[25].

There are also other potential problems that can arise when using concurrency. A *race condition* occurs when the result depends on what affects it first, it is a race between them on who can reach it first[23]. *Locks* can be used to prevent situations as the ones described above by making sure only one thread at a time can access the object[23].
In computer science concurrency occurs when instructions are separated in different *threads*. The architectures can be divided in a taxonomy with four categories including the sequential. They are illustrated in figure 2.4.

- *SISD*(Single-Instruction,Single-Data) is the sequential and is as the name says, single instructions on single data[23].
- *SIMD*(Single-Instruction, Multiple-Data) is applying the same instruction on different data[23]. SIMD is also called vectorisation[23].



(a) *SISD* **(b)** *MISD*

(c) *SIMD* **(d)** *MIMD*

Figure 2.4: *Flynns taxonomy of parallel architectures[23]*

- *MIMD*(Multiple-Instruction, Multiple-Data) applies different instructions on different data[23].
- *MISD*(Multiple-Instruction, Single-Data) performs different operations at the same piece of data, this paradigm is uncommon[23, 26].

2.2.3 Memory, Caching and Optimizations

Various optimizations have been introduce to the CPUs over the years[27, 23]. Some are common and almost all CPUs have them, others are unique to a specific vendor or CPU[23]. Figure 2.5 show the layout of a CPU core. This section explains some of the more common optimizations.

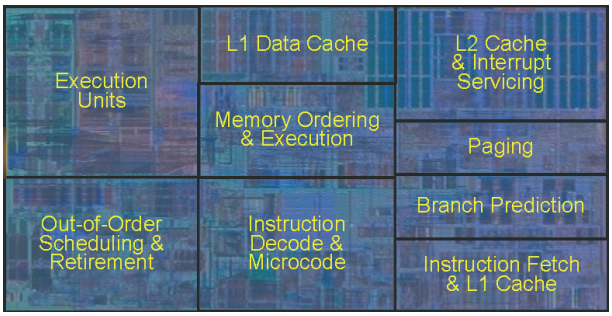


Figure 2.5: *Layout of a CPU core, Intel Nehalem i7, from [28]*

In general retrieving data from memory is relatively slow compared to how fast the processor works[23, 27]. To solve this problem a *cache* was introduced[27]. It stores recently used data in a very small and very fast memory that resides on the CPU chip itself[27]. The caches varies in sizes, the latest ones are around 16-20 mB, for instance Intels i7-5960x with 20 mB [29]. The cached data can then be reused without waiting for the main memory to fetch it. However when the data is not in the cache it costs time to fetch it from the main memory, that is a *cache miss*[27]. Avoiding cache misses is important for the speedup[27]. Modern CPUs commonly have more than one cache, most have three[23]. They are named L1 to L3, with L1 being the one fastest and smallest and L3 the largest and slowest[23].

Modern multi-core processors have several L1 caches, one per core[23]. Commonly each core has a L2 cache as well, however it can be shared in some architectures[23]. The L3 cache is almost always shared between all cores[23]. The multiple caches used causes a problem called *cache coherency*. It is when the data modified in one cache needs to be updated in all the caches in order for them to use the new value[23].

One of the memory optimization methods somewhat related to caches is *prefetching*[23, 27]. Instead of fetching just the data requested by the current operations it also fetches the surrounding data[27]. Even if the data is not needed at the given point in time, the chance that the surrounding data will be used soon is high since it is common to iterate over arrays and similar structures[23].

When a program iterates over an array the first value is a cache miss since it is not in the cache. The following values are prefected and cached as well. However if the array is too long to be prefetched and cached completely and the CPU hits a position not in the cache then a new cache miss happens[27]. Usually matrices are stored sequential in memory[27]. For instance a two dimensional matrix is stored as a one dimensional array. Different programming languages store them differently, either column by column or row by row[27]. Row by row is called *row major* and column by column is *column major*.

Below is an example of how prefectching can have an large impact on performance.

In section 6.2.1 in What Every Programmer Should Know About Memory[27] an example of how much speed that can be gained by optimizing matrix matrix multiplication. The variables have been renamed for clarity. The simple version of it is shown in algorithm 1 and the version with improved cache use is shown in 2.
By transposing the mul2 matrix it becomes more cache friendly by using prefectching of surrounding data the way that mul1 already is[27]. The transpose itself can be eliminated while also making it use the cache better by reading in the correct amount of data, this is the size SM in the code[27]. The optimized version is shown in 2. It took 17.3% of the original time[27].
On top of that by using some additional features of the CPU such as doing multiple instructions at once more speedup is gained increasing it to 9.47% of the original time[27].

Algorithm 1: Basic algorithm for matrix matrix multiplication of two $N \times N$ matrices

Data: Three $N \times N$ matrices, *matrix1*, *matrix2* and *result*

```
for (i = 0; i < N; ++i)
{
    for (j = 0; j< N; ++j)
    {
        for (k = 0;k < N; ++k)
        {
            result[i][j]+ = matrix1[i][k] * matrix2[k][j];
        }
    }
}
```

Algorithm 2: Matrix matrix multiplication optimized

Data: Three $N \times N$ matrices, *matrix1*, *matrix2* and *result*
SM is the number of matrix elements that fits in the cache.

```
for (i = 0; i < N; i += SM)
{
    for (j = 0; j < N; j += SM)
    {
        for (k = 0; k < N; k += SM)
        {
            for (i2 = 0, rresult = &result[i][j], rmatrix1 = &matrix1[i][k];
            i2 < SM; ++i2, rresult += N, rmatrix1 += N)
            {
                for (k2 = 0, rmatrix2 = &matrix2[k][j]; k2 < SM; ++k2,
                rmatrix2 += N)
                {
                    for (j2 = 0; j2 < SM; ++j2)
                    {
                        rresult[j2]+ = rmatrix1[k2] * rmatrix2[j2];
                    }
                }
            }
        }
    }
}
```

If the operation instructions diverge, e.g. the *if* statements, the CPU would have to wait for the results to see what instructions load and do next[27]. This can mean a significant loss of time, however because of an optimizations called *branch prediction* and *speculative execution* the loss of time can be prevented in some cases[27]. A dedicated part of the CPU stores the results from these divergences and when they are encountered again it uses them to make a guess of what do to next[27]. It then loads the instructions and tries to fetch the needed data[27]. The CPU then speculatively executes the instructions, i.e it executes instructions that might not actually be needed[27]. If the guess was correct the results are kept, on the other hand if it was wrong the CPU starts again from the correct path and discards the incorrect results[27].

Sometimes there can be several independent instructions, e.g. when adding two vectors together, this allows optimizations for *instruction level parallelism*[23]. The order of the instructions can be ignored which for instance means that if an instruction needs to wait for some data another instruction that has its data available could be executed while waiting[23]. It is an optimization called *out of order execution*[23]. There are also other possible optimizations[23].
In the example 1 the instructions are independent because they use different variables so they can be executed in any order. If the variables b and c needs to be fetched but e and f are available due to previous instructions then the CPU could execute the second addition first by using out of order execution. However if an operation later depends on the variable a or d then that instruction would have to wait until the additions have been executed.

Example 1: Independent Instructions

Data: Six int variables, $a, .., f$

$a = b + c$
$d = e + f$

Pointer aliasing can prevent the compiler from making certain optimizations, e.g. out of order execution mentioned above. Pointer aliasing is when the same memory area can be accessed using different variables[23]. This becomes a problem if an set of instructions use these variables in the same area of the program. The relative order of operations between these operations then has to be preserved. The compiler does have the information if aliasing occurs or not which means the compiler has to treat all situations where it can occur as if it does[23].

If the integers in example 1 are changed to pointers then aliasing can occur because some of these pointers could point to the same memory. For instance if a and e points to the same memory then the result would be wrong if the second line is executed before the first. This would force the instructions to be executed in order or the result would likely be incorrect.
Some programming languages, e.g. Fortran, disallow some types of aliasing while others, e.g. C/C++, allows aliasing[23]. The compilers for the languages in the second case commonly have an option to disable aliasing. However for

Example 2: Pointer Aliasing

Data: Six int pointers, $a, .., f$

$*a = *b + *c$
$*d = *e + *f$

languages that don't allow aliasing, or if it is disabled, it is then the programmers responsibility to make sure that aliasing does not occur or the results can be wrong[23].

2.2.4 Accelerators, GPU and Xeon Phi

Accelerators are separate parts made to do one task and do it well. They are usually highly specialised and perform badly outside of the tasks they were designed for. The two accelerators explained here is graphical processing unit(GPU) and Intel Xeon Phi[30, 31]. GPUs have mostly been used for games and other graphics related problems however have become more popular for general computing the last 10 years due to their high number of cores[30]. Intel Mic is much more recent with prototypes in 2010 and the first generation released in 2010[31]. It is Intels response to the GPUs and has elements of both GPU and CPU[31]. This section will focus on explaining some parts of the architecture of the GPUs and Xeon Phi. How to use the GPUs are talked about in more detail in section 2.3.

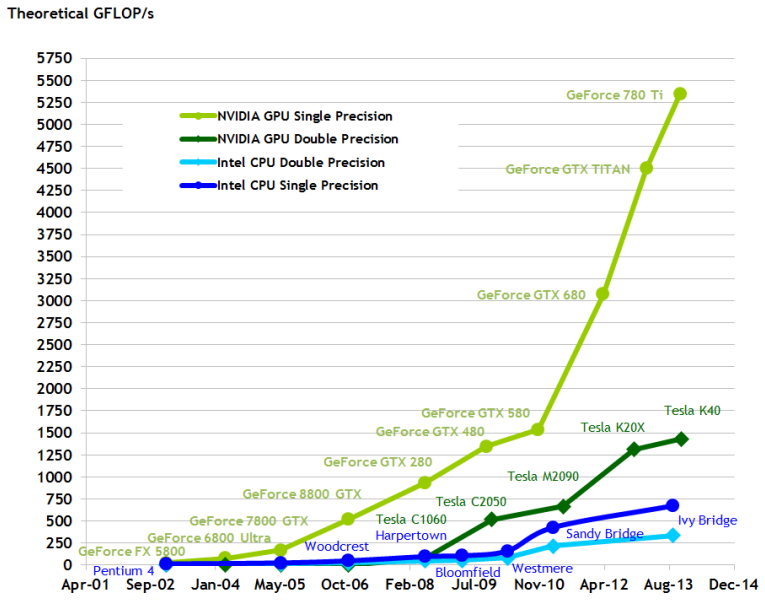


Figure 2.6: *Theoretical throughput of some NVIDIA GPUs and Intel Processors, from[30]. However the chart does not contain a Xeon Phi processor.*

GPUs are made to process images in a fast manner so that games and similar tasks runs smoothly[30]. Because GPUs are made for image processing they

are GPUs have a large number of cores(1000-2000)[30, 33]. Each core is slower than a core in a CPU, however the number of cores makes up for it. The downside is that it's slower than a normal CPU for sequential tasks so the program has to take advantage of the extreme parallelism to get good speed[34]. The memory of the GPU is separate from the computers main memory which means that the data needs to be transferred between the memories which increases the programs complexity[30].

The GPU cores work in a manner that is similar to SIMD[30]. The cores also have fewer optimizations than CPUs, this means that they can then devote a larger portion of the chip to the calculations as seen in figure 2.7. For instance NVIDIAs GPUs lack optimizations such as branch prediction and speculative execution talked about in section 2.2.3 [30]. Their caches are also much smaller than the CPUs[30]. Another inheritance from the image processing is that they are much faster with single precision than with double precision data types[30, 33]. A CPU drops slightly in performance when using double precision however much less than a GPU as seen in figure 2.6. Single and double precision refers to how many bytes are used to store the decimal numbers, fewer bytes means less numerical precision.
Intels answer to the increasing popularity of GPUs for general computing is Xeon Phi[31]. It has elements of GPU architecture however is still more like a CPU. It has separate memory and many cores(50+)[31]. It is MIMD which can be good for codes that diverge a lot, for ease of use however it has the possibly to use more SIMD like structures[31]. It is possible to only use the Xeon Phis memory and avoid the memory transfers that way. The memory is relatively small compared to the main memory so if the program needs a lot of memory transfers are likely needed. The cores of the Xeon Phi lacks some of the normal optimizations, e.g. out of order execution[31].

2.2.4.1 GPU vs Xeon Phi, Why use GPUs for Interaction?

The choice depends largely on the algorithm, performance requirements, etc. For instance if the algorithm contains many points of divergence Xeon Phi is likely faster because Xeon Phi is MIMD while GPU is SIMT. That Xeon Phi is MIMD also means it can be applied to a broader set of problems[31]. A one on one comparison of speed might not the correct choice either because of the different prices of the hardware and power consumption. Speed per power consumption or price can be more relevant.

GPUs are good for interaction algorithms because most methods consider each combination independently and in this case also that LR consists of linear algebra operations. Linear algebra with sufficiently large matrices and vectors suits well with the GPUs SIMT architecture[36, 30]. Several studies got high gains from implementing their algorithms on GPU[8, 37, 38, 39, 40, 41]. However most studies had CPU versions that were not parallel so it is likely that the gains would have been less if they had compared to an optimized and parallel CPU version. One study made a comparison of their algorithm between using a CPU cluster and using a GPU[42]. They found that 16 CPU nodes had the same performance as a single GTX 280 card[42].

2.2.5 Clusters

Clusters are collections of computers that can work together in several ways and are so tightly connected that they can usually be viewed as a single system. They communicate over a shared network but have separate memory and processors. Storage is usually shared. A computer in a cluster is called a *node*. [43, 23]

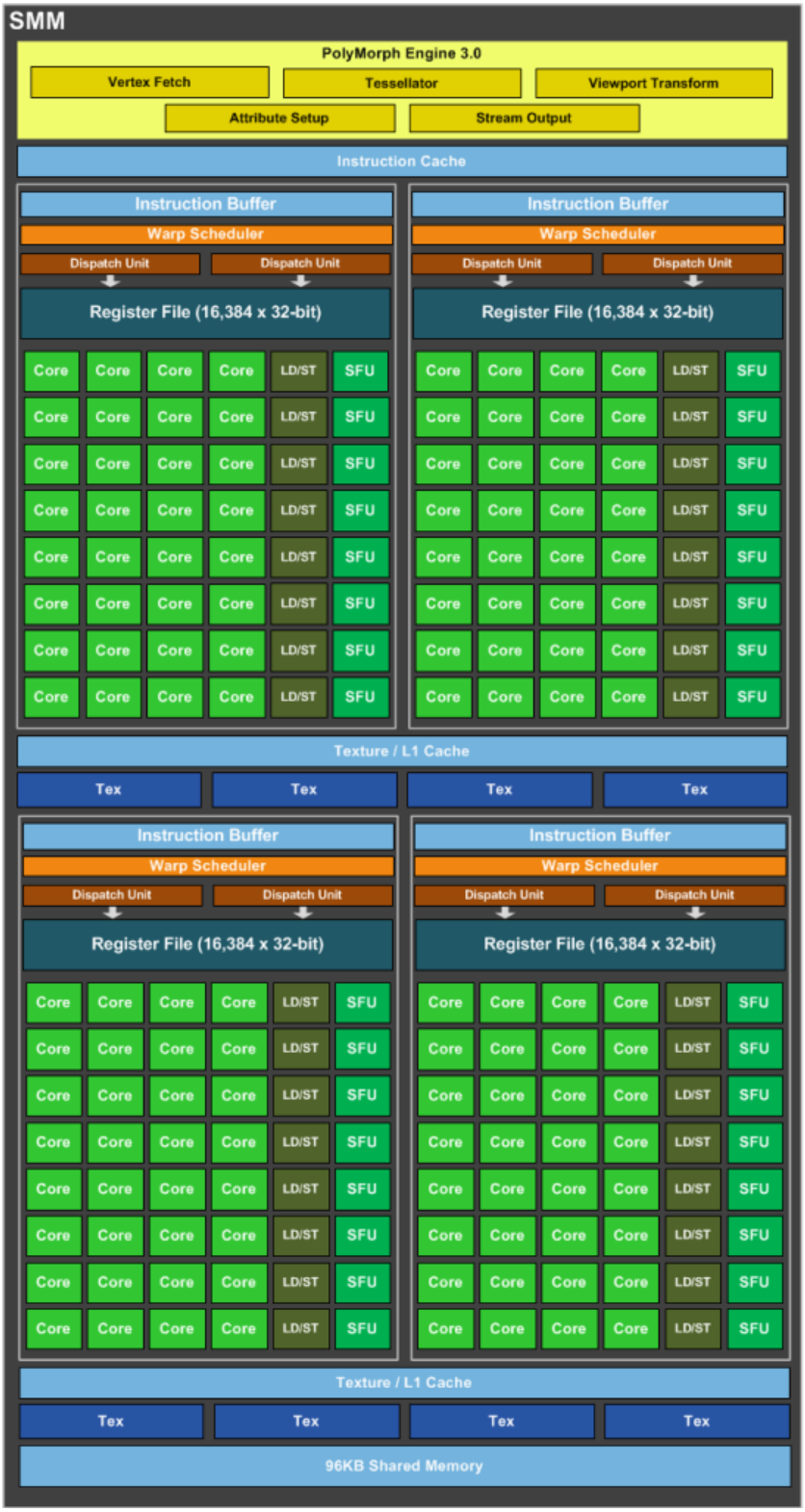


Figure 2.7: *Layout of the Maxwell architecture, from [32]*

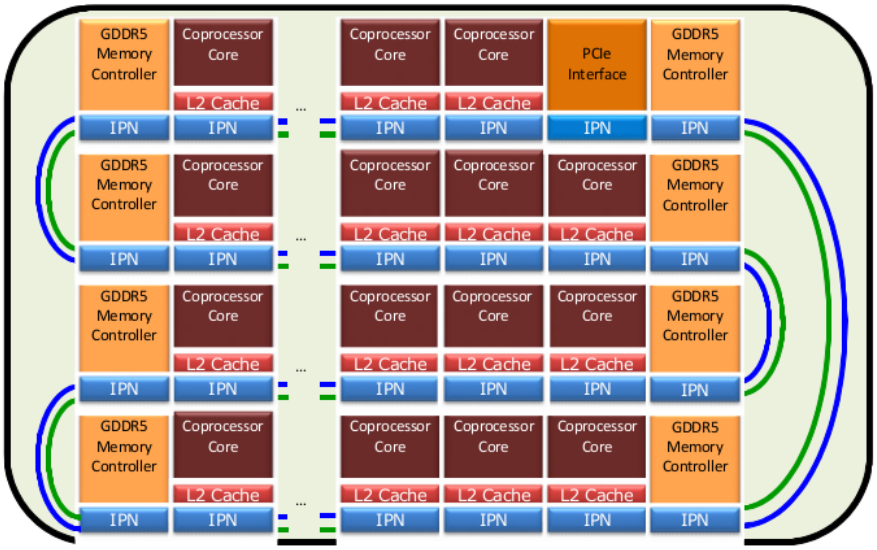


Figure 2.8: *Knights Corner silicon layout, from[35]*

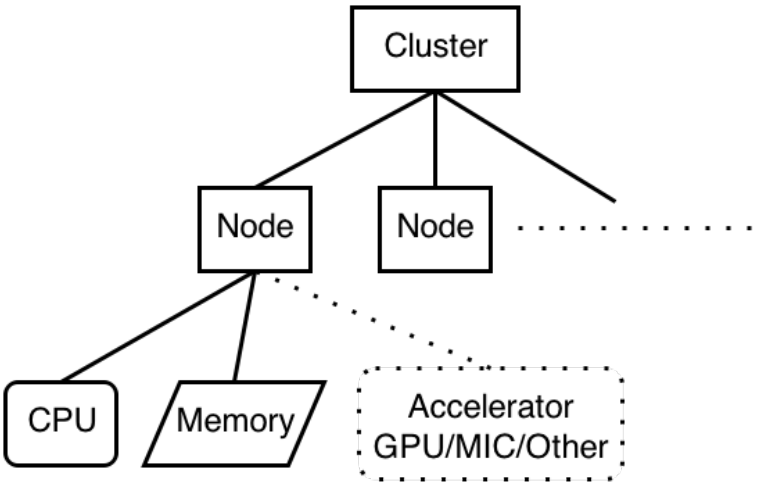


Figure 2.9: *Basic overview of a cluster*

2.3 CUDA programming model

This section is about GPU programming in more detail using NVIDIAs CUDA(Compute Unified Device Architecture)[30]. CUDA extends C and C++ with some additional functionality that can be used to perform operations on NVIDAs GPUs. It provides a separate compiler to compile the GPU code called nvcc[30]. Different GPUs support different CUDA functions, each NVIDIA GPU has a value called computational capability. The higher the value the more features of CUDA are supported on that GPU. Some properties also vary between the different underlying architectures. This means that it is important to know what kind of capability the GPU to be used has so that the program does not need features that are not there. The things described here might not apply to GPUs with compute capability below 3. There are GPUs made specifically for calculations rather than games. This section is an summary of how CUDA works from the CUDA programming guide[30] and CUDA best practices guide[34].

In CUDA the GPU is called *device* and systems CPU and memory is the *host*. To perform operations on the device a type of function called *kernel* is used. A kernel is works mostly as a normal C/C++ function with the addition of some specifiers to provided options to set number of threads and so on. Example 3 is an example of a simple kernel definition and example 4 shows how it can be called from the host code. The kernel add each element of the arrays A and B storing it in C. Each thread performs on addition and knows why elements to use based on it's id. The call to the kernel is made by giving three arrays and the number of threads to be used as N. In this case N needs to be the length of the arrays.

The ___global___ keyword in front of the function declaration tells nvcc that it is a kernel. The $<<< N, M >>>$ specifier tells the compiler how many N blocks and how many M threads per block that the kernel should use[30]. A block is a group of threads and shares some memory and resources. The blocks can be executed in any order so they need to be completely independent from each other but variables can be shared among threads inside a block[30]. This allows the program to scale to different GPUs as shown in figure 2.10. The blocks are organized in a one, two or three dimensional *grid*. These can be accessed by each thread so it knows which grid it is in as illustrated in figure 2.11. This can be used to make it easier to assign the threads to the correct bit of the calculation. For instance using a two dimensional grid is good for matrices[30, 34].

Example 3: Example of a declaration of a simple kernel

Data: Vectors A, B, C as C-style arrays

```
___global___ void Add(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

Example 4: Host code to call the kernel in example 3 with N threads

Data: Vectors A, B, C of length N as C-style arrays

Add$<<<1,N>>>(A, B, C)$;

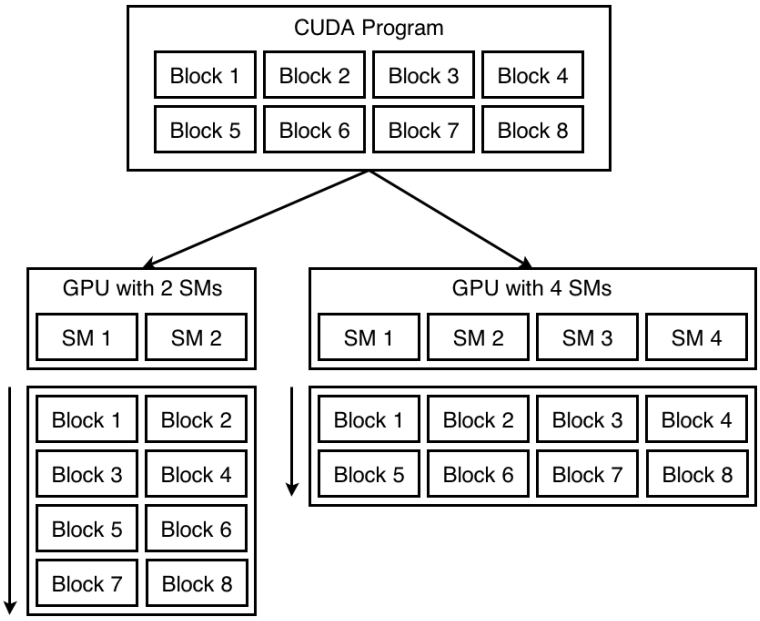


Figure 2.10: *Blocks execution depending on the number of streaming multiprocessors*

The streaming multiprocessors(SM) is the hardware that handles the execution of these blocks and can execute hundreds of threads concurrently. It uses *SIMT*(Single-Instruction, Multiple-Thread) which is similar to SIMD described earlier in section 2.2.2. SIMT works mostly as SIMD except that it can act as MIMD on collections of threads called *warps*. Each warp consists of 32 threads. When an SM gets a block it is split into warps that are assigned to warp schedulers. Each warp scheduler gives one instruction to a warp so full efficiency is achieved when all the 32 threads perform the same instruction. If there is any divergence it has to disable unrelated threads, so divergence can be costly. However different groups of warps are on different warps schedulers so can diverge without problem.[30]

2.3.1 Device Memory

The GPUs memory is physically on a different device separate from the computers main memory which means that they have separate memory spaces. An object in one memory is not accessible in the other memory. The computers main memory is the *host memory* and the GPUs memory is the *device memory*. Since they are separate data has to be transferred to the device memory and this is done by explicit calls to transfer sections of the hosts memory.[30]

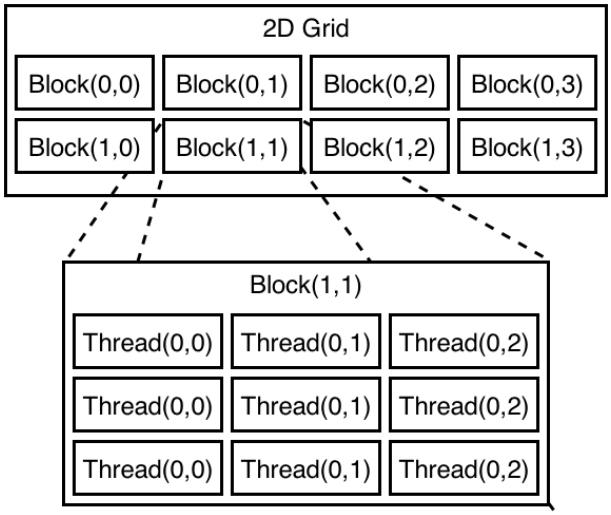


Figure 2.11: *2D grid of blocks*

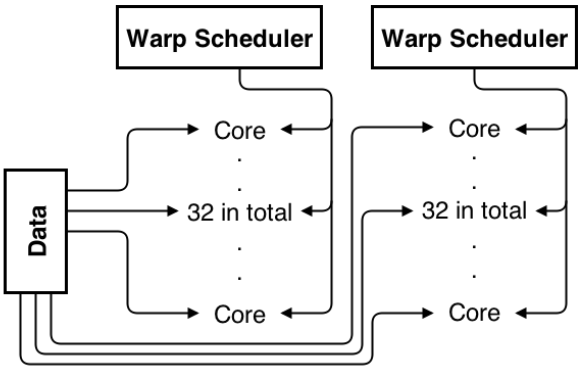


Figure 2.12: *SIMT architecture*

The GPU also have several different types of memory[30]. Correct usage can give increased speed[30, 34].

- Register memory is located on the multiprocessor and usually costs zero cycles to access. The multiprocessor splits the available registers over its threads so if there are many threads that uses many variables not all of them will fit in the register. This is why a program sometimes can be faster with lower number of threads.[30]
- Global memory is the main memory of the GPU and is accessible from all threads and blocks. However it is relatively slow to access.[30]
- Shared memory is shared inside a block and is faster than global memory. However it is limited in size.[30]
- Constant memory is small however it is read only which enables some optimizations. It is best used for small variables that all threads access.[30]

- Local memory is tied to the threads scope, however it still resides off-chip so it has the same access time as global memory.[30]
- Texture memory is read only and can be faster to access than global memory in some situations. This was more important in older GPUs when global memory was not cached.[41, 30]
- Read-only cache is available on GPUs based on Kepler architecture and uses the same cache as the texture memory. The data as to be read only each multiprocessor can have up to 48kb of space depending on GPU.[44]

2.3.2 Streams

The kernel calls can be made on a *stream*. The easiest way to think of the stream is as queue. The kernels on a stream will be executed in the order they are made but kernels from different streams can be executed in any order given there is enough computational resources. In this way it is possible to execute up to 32 concurrent kernels depending on GPU.[30]

This is not entirely true for some GPU architectures because they only have one queue for the kernels. Because of this independent kernels from different streams can block each other. If we have a group of kernels and issue them on stream one first and then on stream two all the kernels in stream two will be stuck in the queue waiting for the kernels in stream one to finish. This is because the kernels queue only looks at the first kernels to see if they can be executed. The second kernel in the first stream then blocks the queue from moving forward so it will not see the kernels on stream two that could be executed. Newer architectures after Kepler on the other hand has several queues so they don't have this problem.[30, 34, 44]

Streams are also important for asynchronous transfers. These transfers are executed on a stream and just as kernels gets executed after the previous kernels on the same stream is done. The advantage is that other streams can do calculations as normal while the transfer happens. This can hide the time for transfers completely in some situations as shown in 2.14. However the host memory has to be pinned, pinned memory means that the operative system can not page that memory. Paging is that the operative system stores a part of the memory in another area, usually the disk memory, to save space in the memory. Too much pinned memory can slow down the computer.[30, 45]

However there is problem that asynchronous transfers can cause depending on the GPUs architecture[45, 46]. Some older GPUs only have one copy engine while newer have one for each direction, one to the GPU and one from the GPU[45, 46]. This can affect how the calls should be ordered and using the wrong one can make the performance worse than without using asynchronous transfers[45, 46].
There is an example of this in [46] which illustrates the problem. They have four versions of the same code, a sequential transfer version and three asynchronous transfer versions. Two different GPUs were used, one had one copy engine the other had two. In the asynchronous the data is split over four streams coloured differently in the figure 2.14. Version 1 initiates the calls by looping over the

streams one by one and doing the transfer and kernel calls on that stream before moving on to the next[46]. Version 2 makes all the host to device transfer calls for all streams first, then the kernels and then the device to host transfer call[46]. Version 3 is the same as version 2 but with a dummy even after each kernel[46]. The figure 2.14 shows how the transfers and kernels are executed on the GPU.

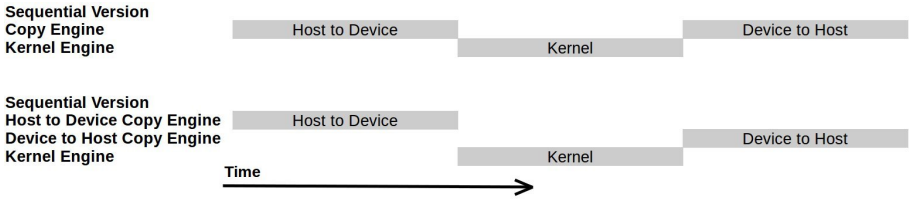


Figure 2.13: *Sequential versions*

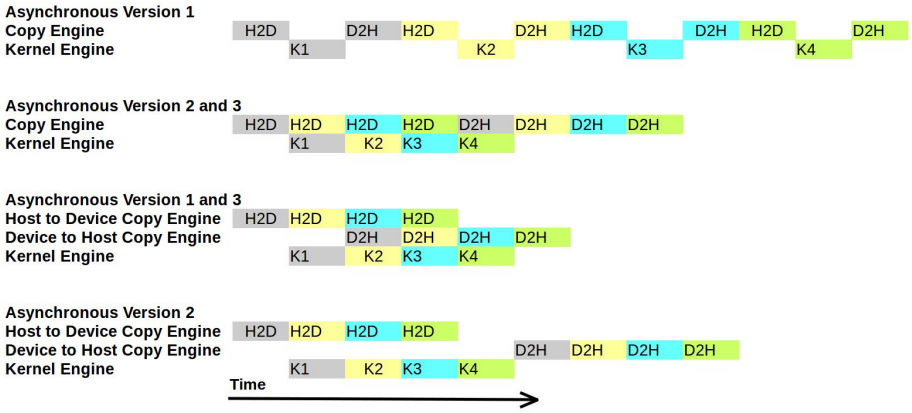


Figure 2.14: *Asynchronous versions. D2H=device to host transfer. H2D=host to device transfer. Each colour represents a stream.*

2.3.3 Efficient CUDA

One of the main criticism against GPUs for general computing purposes is that it is hard to get good performance because it requires good knowledge about details of the GPU architecture, especially the memory architecture. This section is a summary of things that can be good to consider for CUDA programs from the CUDA programming guide[30], CUDA best practices guide [34] and a similar list as this one, however slightly outdated now since it is from 2010, from a master thesis about GPU in GWAS[41].

Maximize parallelism

Structure the program and the algorithm in such a way that it is as parallel as possible and overlap the serial parts on CPU with calculations on the GPU.[41, 30]

Minimize transfers between host and device

Moving data between host and device is expensive and should be avoided

if possible. It can be better to run serial parts on the GPU rather than moving the data to the host to do the calculation on the CPU. The bandwidth between host and device is one of the large performance bottlenecks. This can be a problem when the data is to large to fit in the relatively small GPU dram.[30, 34]

Find the optimal number of blocks and threads

There are many things affected by the number of blocks and threads so they should be considered carefully. It is a good idea to parametrize them so that they can be changed for future hardware and varied for optimization. NVIDIA has an occupancy calculator which can be helpful in determining the optimal numbers, however high occupancy does not mean high performance.[30, 34]

The number of blocks should be larger than the number of multiprocessors so that all multiprocessors have at least one block to execute. Having two blocks or more per multiprocessor can be good so that there are blocks that aren't waiting for a ___syncthreads() that can be executed. However this is not always possible due to shared memory usage and similar.[34]

The number of threads per block should be a multiplier of 32 but minimum 64. It's also important to remember that multiple concurrent blocks can reside on the same multiprocessor. Too large number of threads in a block and parts of the multiprocessor might be idle since there aren't a block small enough to use those threads. Between 128 and 256 threads is a good place to start.[34]

Use streams and asynchronous transfers

By using streams it is possible to overlap memory transfers with calculations as mentioned before. This means that the data for the next batch can be transferred while the current batch is calculated and when it is done it can start calculating on the next batch directly after the current one is done. This can hide the time for transfers completely in some situations. Depending on the time the transfers take versus the time the calculations take this can give significant speedup.[30, 45, 44]

Use the correct memory type and caches

Correct use of caches and memory is important for both CPU[27] and GPU. However it is more complicated on GPU since the caches are smaller and there are several types of memory as mentioned before[30, 34].

Avoid divergence

Each thread in a warp executes the same instruction at the same time so if some of threads diverge the rest will be ideal until they are at the same instruction again. This means it is important to use control structures such as if statements carefully to prevent threads from idling.[30, 34]

Avoid memory bank conflicts when using shared memory

Shared memory is divided into equally-sized memory modules called banks that can be accessed at the same time for higher bandwidth. Bank conflicts occur when separate threads access the same bank. On some GPUs it is

fine if all threads access the same bank. Bank conflicts are split into as many conflict-free requests as needed.[30, 34]

Use existing libraries

Instead of writing everything from scratch it is usually a good idea to use already existing libraries. Especially when performance is important and most task are non trivial on GPUs so using an already optimized library is a good idea. Some of the most popular libraries for CUDA are:

- CUBLAS: BLAS implementation for CUDA. BLAS and LAPACK is a standard for a library that provides highly optimized functions for linear algebra.[36]
- CULAtools: BLAS and LAPACK implementation for CUDA for both dense and sparse matrices[47]
- MAGMA: BLAS and LAPACK implementation among other things that can distribute the work on both CPU and GPU[48]
- Thrust: Template based library that tries to emulate C++ standard library[49]

Avoid slow instructions

There are some instructions that can be slow and should be avoided if possible, for instance type conversion, integer division and modulo. If a function is called with a floating point number that might be used as a double and require a conversion. By putting an f at the end of the number it is told to be single precision float, for instance 0.5f. In some cases it is possible to use bitwise operations instead which is faster.[30, 34]

Restricted pointers can give increased performance

Aliasing can be a problem as mentioned earlier. By using the ___restrict___ keyword on pointers the compiler can be told that no aliasing will occur, however it is up to the programmer to make sure that is the case or there might be unexpected results. Not using aliasing reduces the number of memory accesses the CPU needs to make. However it increases register pressure so it can have an negative effect on performance.[30]

Use fast math functions if precision isn't needed

There are three versions of the math functions. The double precision version is func() while the single precision function is funcf(). There is third faster but less accurate version __funcf(). The option -use_fast_math makes the compiler change all the funcf() to __funcf().[34]

Instruction level parallelism can increase speed

Just as for CPUs the GPUs can take advantage of instruction level parallelism. By unrolling loops this can give 2x the speed relatively simple[50].

2.4 Software Design Background

How to design software so it can be maintained over time has been a problem for a long time. Object oriented languages such as C and later Java and C++ arose because of problems with maintaining software. Badly organized and written code will cost productivity in the future when bugs and other problems stack up because of earlier mistakes. Fixing those bugs are likely to time consuming because it can be hard to find where they originated from. All code gather problems overtime, however a well designed system will degenerate significantly slower than one that was designed carelessly. The code will also be read by ourselves and others later which means readability is important if a future reader is to understand the code and find the parts they are interested in. [51, 52]

This is also important in science where others might wish to use the tools or repeat the experiments. A recent study[53] found that the repeatability in computer science is low. They used the term reproducibility however repeatability is a more fitting term because they didn't try to reproduce the results from scratch, they tried to repeat it using the same code. Of 513 articles they received the code from 231, 108 failed to build, 21 failed to run properly[53]. The remaing 102 compiled and ran, however they did not check if the results of the program was correct[53].

There are ways to design the program and code in such a way that it is easier to read and maintain. Most of the things described here goes under the development technique called agile development and the concept of clean code and is a summary of some concepts from the book Clean Code[51]. SOLID is an acronym for five principles for object oriented programming and design[51]. When used together they are intended to make programs that are easy to maintain and extend[51]. As already mentioned readability is important. Correct names will make the code explain itself without other documentation, the code itself is the documentation[51].

Initial	Principal	Concept
S	Single responsibility principle	A class should only have a single responsibility
O	Open/closed principle	A class should be open for extension but closed for modification
L	Liskov substitution principle	If S is a subtype of T, then objects of type T may be replaced by objects of type S without without altering any of the desirable properties of the program(e.g. work performed)
I	Interface segregation principle	Use several smaller and more specific interfaces instead of one large
D	Dependency inversion principle	Depend on abstractions(e.g. interfaces) not details

Table 2.4: *The five SOLID principles, from [51]*

Dependency injection is one way to implement dependency inversion principle[51]. Injection is passing the dependency to the dependent object. This is used instead of allowing the dependent object to construct or find the dependency.

2.4.1 Unit Tests and Mocks

Unit testing involves testing the program in small units in isolation. Testing in isolation means that the test should only depend on the part of the program that is tested. If a part of the program is not working properly only its related tests should fail, not other tests for code that depends on it but otherwise works properly. This makes it easier to find errors when they do occur since the tests will pinpoint the unit which does not work. *Integration testing* is used to test several units together to find possible errors
[51]

It can be easy to denote test code as less important than the main code but they should be treated as equally important. Tests should also not be an inconvenience to use so they should be easy to run, take reasonable amount of time to complete and not require any outside interpretation whether they failed or not.[51]

Mocking is to replace a real object with a fake object called a *mock* that for the code is indistinguishable from the real object. This allows one to create situations and test with more control and without depending on the real objects code. The second is important for unit tests since it enables one to test units that normally depend on others in isolation. In the first case it is useful in situations such as when one wants to test a class handling of a rare failure. Such failures can be hard and time consuming to induce. It is then easier to use a mock object that behaves like the failure has occurred. However mocking requires that the code for the class doesn't create the object itself since there is no way to replace the object with the mock. This is one of the reasons why dependency injection should be use.[51]

2.4.2 Design Patterns

A *design pattern* in software design is a reusable general solution to a common recurring problem in a given context. It is templates and structures to solve the problem, however it is not code. However they are partially dependent on the programming language since different languages have different features and limitations.[52]

Consumer producer is a concurrency pattern for when there are a number of consumers/workers and producers. They share a common queue for products. The producer generates some product and put it into the queue, while the the consumers consume the products. The problem is that the producers should not add to an non empty slot and that each product should only be consumed once.[51, 52]

Configuration pattern classes are used to reduce the number of parameters to

functions. Instead of having a number of parameters to a function the parameters are put in a Configuration class. The Configuration class has two functions for each parameter, a **set** function and a **get** function. The **set** is used to set the parameter while **get** is used to get the parameter.

2.4.3 Version Control Software

Backups for data, code, text and so on is important if something happens. One way to backup text and code projects is to use a version control software and is usually an important tool in developing[54]. For this project Git was used and the source code can be found at `https://github.com/Berjiz/CuEira`. Version control software allows several developers to share a common *repository* which allows them to work on different parts at the same time, it could even be in the same file[54]. However changes at the same place causes *merge conflicts* which usually needs to be solved manually[54]. A version control software keeps track of all the changes made so if a part later turns out to be wrong that part can reverted to an older correct version[54]. Version control software also have many other features[54].

2.5 Performance Measures

An important part in creating fast and efficient programs is to know how fast the program is under certain conditions and which parts of the program are slow[23]. For instance the speed could suddenly drop when too many threads, there might be a bottleneck in the communication, and so on.

There are two ways to measure how long a program takes to execute[23]. Wall clock time is how long real life time the program took. The other is to measure the number of processor cycles spent. A parallel program will have shorter execution time than it is serial version however it will likely have spent more processor cycles due to overhead from communication and initialization of the threads. These two measures are useful for different kinds of comparisons. Wall clock time is better for overall performance while number of cycles is useful for comparing different algorithms[23, 34].

Speed up is a measure of how much faster then program is with a certain number of threads compared to the serial version. It's defined as[23]

$$S(p) = \frac{T(1)}{T(p)}$$

Where $T(1)$ is execution time of serial program and $T(p)$ is execution time of parallel program with p threads. Linear speedup is when S(p)=p[23].

Efficiency reflects how efficient the program is using p threads. Linear speed has efficiency 1. It's defined as[23]

$$E(p) = \frac{S(p)}{p} = \frac{T(1)}{pT(p)}$$

Strong scaling refers to how the program handles a fixed problem size and increased number of processors[23]. An program with strong scaling has linear speedup[23]. Weak scaling refers to the execution time of the program when there is a fixed problem size *per processor* and the number of processors is increased[23, 34].

It can be a good idea to plot these measures while varying p, this can show when a bottleneck occurs. Looking at the measures at node level can also be useful to get an idea of how increased number of nodes and therefore increased communication over the network affects the performance.

2.5.1 Amdahl's Law and Gustafson's Law

Amdahl's Law is used to find the expected speedup of a system when parts of it are made concurrent. Simply it says that as the number of processors increases the parts that aren't parallel will start taking up more and more of the wall clock time and that the speedup for adding more processors will decrease as more and more processors are added and more time is spent relatively on the non parallel parts[23]. It's closely related to strong scaling.[34, 55]

It says that the expected speedup with F fraction of the code parallel and p threads is[23]

$$S(p) = \frac{1}{(1-F) + \frac{F}{p}(1-F)}$$

As the number of threads grow towards infinity $S(p)$ converges on $\frac{1}{1-F}$. If we have 90% of a code parallel then even with infinite number of threads we won't get a better speedup than ten.[55]

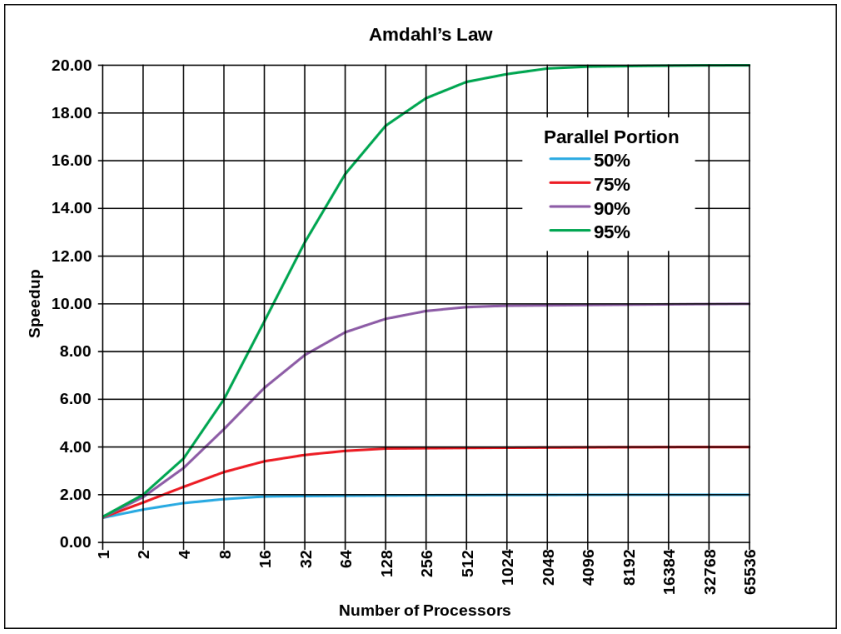


Figure 2.15: *Illustration of Amdahl's Law. Wikipedia Commons*

There are limitation to Amdahl's Law since it makes a couple of assumptions.

- The number of executing threads remain constant over the course of the program.
- The parallel portion has perfect speedup. Often not true due to shared resources,eg caches, memory bandwidth, and shared data.
- The parallel portion has infinite scaling, not true due to similar limits as above. More threads will not increase performance after a while or might even decrease it.

- There is no overhead for creation and destruction of threads.
- The length of the serial portion is independent of the number of threads. Often the serial work is to divided the work to the threads, this work will obviously increase as the number of threads go up. More threads can also lead to more communication overhead.
- The serial portion can't be overlapped by the parallel parts. For instance with producer consumer type pattern the consumer could be strictly serial but the time it takes could be overlapped by the parallel producers.

This means it is most accurate with programs that are of the fork-join type, e.g. both serial and parallel parts[56].

Gustafson's Law is closely related to Amdahl's Law and can in some ways be more accurate than Amdahl's Law[56]. Gustafson's Law makes similar assumptions as Amdahl's Law however it also makes two additional statements. It states that problems tends to expand when provided with more computational power, e.g. increased precision by reducing grid size for simulations, higher frame rate for graphics and so on[56, 23]. The second is that the parallel portion of the program tends to expand faster than the serial part, e.g. for matrix multiplication the initialization scales linearly with the matrix size while the multiplication itself scales as $O(n^3)$[56, 23]. The former means that it is closely related to weak scaling[23]. So in a way it says that the execution time remains constant rather than the amount of data. More precise it says that the expected speedup with p threads and F fraction of the code that is parallel is[56, 34, 23]

$$S(p) = p + (1 - F)(1 - p)$$

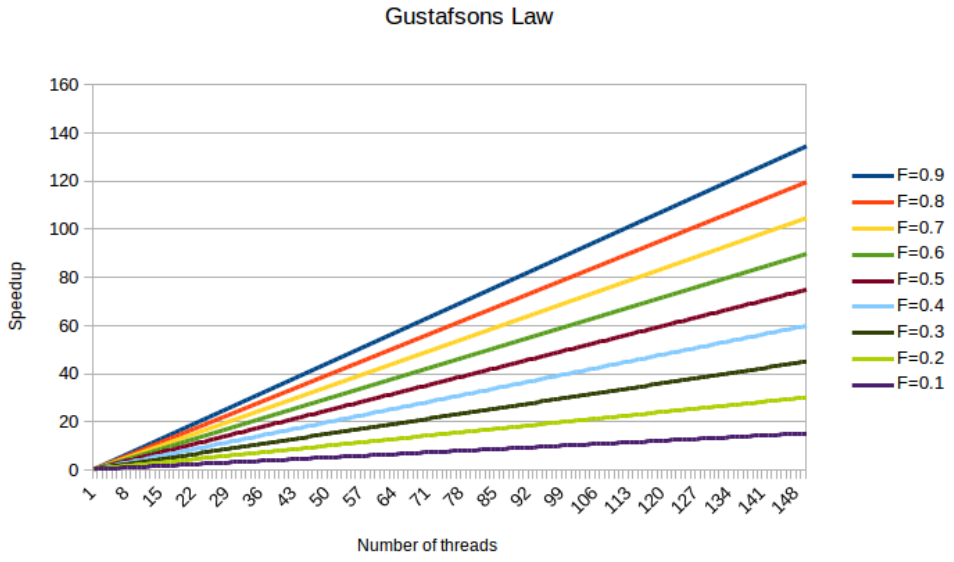


Figure 2.16: *Illustration of Gustafson's Law*

2.5.2 Profilers

There are applications called profilers that are made to assess the programs performance and resource consumption. They calculate some of the measures mentioned earlier and they also check hardware usage and how much time the program spends at various parts of the program. This is very useful for finding bottlenecks and other problems in the program. It does not matter if the algorithm is super fast if all the data is stuck in network transfers. The profilers can be hardware dependent so the manufactures usually provided them for their products.[23, 34]

Nsight[57] is a combined profiler and integrated development environment(IDE) based on either Visual Studio or Eclipse. For this thesis Nsight Eclipse edition was used. It works as the usual Eclipse but has added functionality for CUDA and CUDA profiling[57].

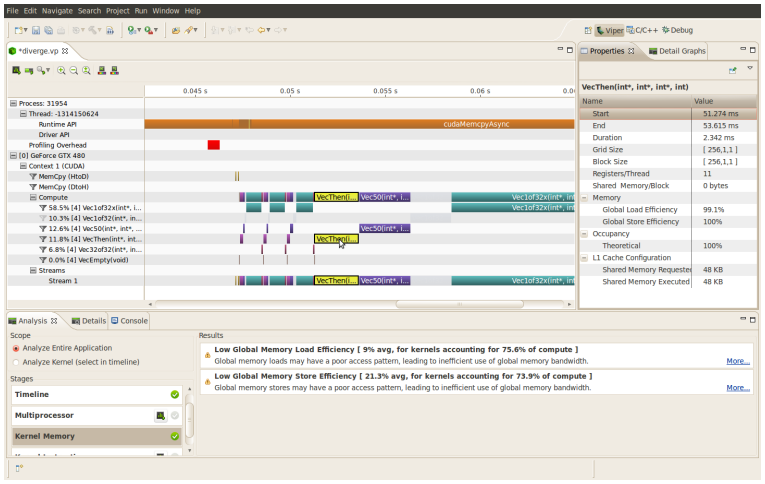


Figure 2.17: *Screenshot of Nsight Eclipse Editions profiling section, from [57]*

2.6 Algorithms

There are many algorithms and programs proposed for searching for interaction, most have focused on gene-gene interaction as already mentioned[2]. One of the challenges of gene-environment interaction is that environmental factors can be of any variable type(i.e. binary, continuous, categorical) which creates problems in various ways[2]. Gene-gene interaction tools can sometimes be used to find gene-environment interaction, however they usually require the variables to be binary or have other problems since they weren't designed for environmental interaction[2].

Both groups of computers(i.e. clusters) using regular processors[58] and graphic processors[8, 37, 38, 39, 40, 41] have been used for GWAS. Graphic processors for computing have become more popular in the last ten years. They have been a popular choice for some GWAS methods because each combination of variables can commonly be considered independently from the others. More about graphic processors in section 2.2.4 and why the are good for GWAS in section 2.2.4.1.

The methods can be roughly classified into four categories, exhaustive, stochastic, machine learning/data mining and stepwise[10].

Exhaustive search is the most direct approach, it compares all combinations of the SNPs in the dataset. Exhaustive search methods will not miss a significant combination because it didn't consider that specific combination. However it also means that they can be slow since they will spend time on combinations other methods would skip completely. Multifactor-Dimensionality Reduction(MDR)[59] and BOOST[60] are two examples of this type of algorithm.

Stochastic methods uses random sampling to iterate through the data. BEAM[61] is one example and it uses Markov Chain Monte Carlo(MCMC) method.

Data Mining and *Machine Learning* are methods that try to learn patterns from data and tries to generalize it. MDR[59] is a type of data mining method and is among the most common methods used in GWAS. See section 2.6.2 for more details.

Stepwise methods uses a filtering stage and a search stage. At the filtering stage uninteresting combinations are filtered out by using some exhaustive method. The other SNPs are the examined more carefully in the search stage. BOOST[60] is an example which uses succinct data structures and a likelihood ratio test to filter the data before applying log-linear models.

2.6.1 Logistic Regression

One way to model the contingency tables is by using *logistic regression*. Logistic regression is a type of linear regression model for classification that models a latent probability for the outcomes. The outcomes are binary, however the method can be extended to multiple outcomes. In this work we will only consider them as binary. Logistic regression transforms the probability by using the *logit* transformation. The logit transformation with probability π is [17]

$$\log(\frac{\pi}{1-\pi}) \tag{2.11}$$

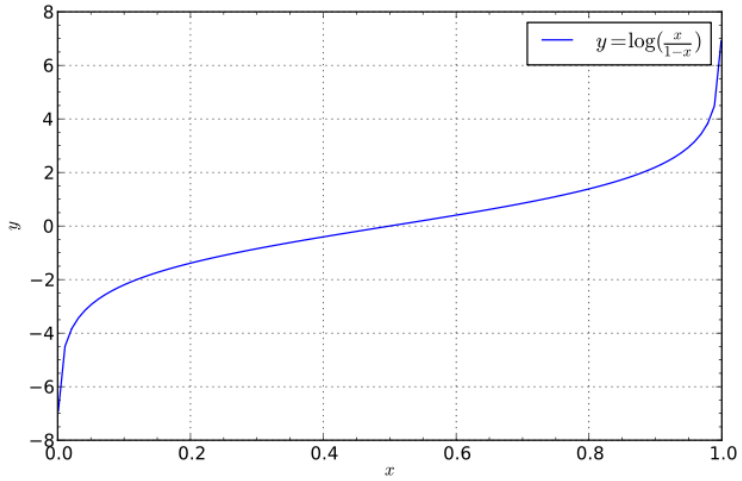


Figure 2.18: *Graph of the logit transformation. Wikipedia Commons*

The probability with a set of predictor variables X is $\pi(X) = P(Y = 1)$. The linear regression model with n predictors $X = (x_1, x_2, ..., x_n)$, coefficients $\beta = (\beta_1, \beta_2, ..., \beta_n)$ and by using the logit transformation is then[17]

$$logit[\pi(X)] = \alpha + \beta X \tag{2.12}$$

By moving the logit to the right side of the equation we get the model of the probability[17]

$$\pi(X) = \frac{e^{\alpha+\beta X}}{1 + e^{\alpha+\beta X}} \tag{2.13}$$

The logit, equation 2.11, also happens to be the log of the odds(equation 2.5)[17]. By exponentiating both sides of equation 2.12 it shows that the odds is [17]

$$e^{logit[\pi(X)]} = \frac{\pi(X)}{1 - \pi(X)} = e^{\alpha+\beta X} = \Omega \tag{2.14}$$

This means that $\exp \beta$ is the odds ratio since the odds increase by $\exp \beta$ for each unit increase of X[17]. It can also been seen by taking the ratio of the odds using equation 2.6 and with $X = x + 1$ and $X = x$

$$\theta = \frac{e^{\alpha+\beta(x+1)}}{e^{\alpha+\beta x}} = e^{\alpha+\beta(x+1)-\alpha-\beta x} = e^{\beta} \tag{2.15}$$

Finding the β coefficients are done in a similar way as with other linear regression models since they all are generalized linear models [17]. It's usually done using maximum likelihood(ML), via Newtons method[17, 12]. It's an iterative method which means it can be relatively slow compared to non iterative methods. The pseudo code for the algorithm using Newtons method can be found in algorithm 3 [12]. Line 13 is sometimes inside the loop, however since it is not needed for the actual iteration calculating it for every loop wastes time.

Algorithm 3: Logistic regression using maximum likelihood and Newtons method

Data: N number of data points
M number of variables, excluding the intercept
$\boldsymbol{X}$ is an $N \times M$ matrix that contains the variables
$\boldsymbol{Y}$ is the outcomes with length N
β has length M and contains the initial values of β
Note: $*$ is element by element multiplication

1 $\boldsymbol{X} \longleftarrow \begin{pmatrix} 1 & \\ \vdots & \boldsymbol{X} \\ 1 & \end{pmatrix}$

2 $\beta \longleftarrow \begin{pmatrix} 0 \\ \beta \end{pmatrix}$

3 $iter \longleftarrow 0$
4 $diff \longleftarrow 1$

5 **while** $iter < max\_iter$ ***and*** $diff > threshold$ **do**
6 $\quad \beta_{old} \longleftarrow \beta$
7 $\quad p \longleftarrow \frac{e^{\boldsymbol{X} \cdot \beta}}{1+e^{\boldsymbol{X} \cdot \beta}}$
8 $\quad s \longleftarrow \boldsymbol{X^T} \cdot (\boldsymbol{Y} - p)$
9 $\quad \boldsymbol{J} \longleftarrow (\boldsymbol{X^T} \cdot (p * (1-p))) \cdot \boldsymbol{X}$
10 $\quad \beta \longleftarrow \beta_{old} + \boldsymbol{J}^{-1} \cdot s$
11 $\quad diff \longleftarrow \sum |\beta - \beta_{old}|$
12 $\quad iter \longleftarrow iter + 1$

13 $log\ likelihood \longleftarrow \sum(\boldsymbol{Y} * lnp + (1 - \boldsymbol{Y}) * ln(1-p))$

2.6.1.1 Matrix Inverse and Matrix Decomposition

The inverse of the *information matrix*, **J**, on line 10 in algorithm 3 is generally not possible to do with normal matrix inversion because it is not defined for a general matrix[62]. However the *pseudoinverse* is defined for a general matrix, it is denoted as A^+ for the matrix A[62, 63]. There are several types of pseudoinverse, one of the more common is the *Moore-Penrose pseudoinverse* where A^+ is the matrix that satisfies equations 2.16 to 2.19[62].

$$AA^+A = A \tag{2.16}$$

$$A^+AA^+ = A^+ \tag{2.17}$$

$$(AA^+)^T = AA^+ \tag{2.18}$$

$$(A^+A)^T = A^+A \tag{2.19}$$

Matrix decomposition, also called *matrix factorisation*, are methods to factorize a matrix into products of matrices[63]. Some of the decomposition methods can be used to find the pseudoinverse. *Singular value decomposition*(SVD) is one of them and works for general matrices[63]. It's factorization of a $n \times m$ matrix A is shown in equation 2.20[63].

$$A = U\Sigma V^T \tag{2.20}$$

Where Σ is a diagonal matrix with non negative numbers, its diagonal values are the *singular values* of A[63]. Using SVD the pseudoinverse is shown in equation 2.21[63]. Σ^+ is the pseudoinverse of Σ. This is done by inverting each non zero element of the diagonal and transposing the matrix[63].

$$A^+ = V\Sigma^+U^T \tag{2.21}$$

2.6.2 Data Mining and Machine Learning Approaches

Approaches based on Data Mining and Machine Learning have been a popular choice for GWAS. MDR[59] and Random Forest(RF)[64] are among the most common ones[2, 1]. There are other methods as well such as clustering approaches [10]. Most of them are used for screening the data for possible interactions[2, 1].

Their biggest advantage is that they are usually non-parametric and designed with high dimensional data in mind. However they are prone to overfitting and the usual way to try to prevent that is to use cross validation and sometimes permutation tests. It means that even if the method itself is fast it is repeated so many times that the whole algorithm can be slow in the end.[1]

2.6.2.1 Multifactor-Dimensionality Reduction

MDR is a method that reduces the number of dimensions(i.e. variable) by combining several dimensions into one. In GWAS it combines the variables that are suspected to interact. This new variable is then compared against the outcome. If the new variables predictability of the outcome is high enough then the variables that were combined are considered to interact. This process is usually repeated on all pair combinations of variables.

The reduction from n dimensions is done by calculating the ratio of cases versus controls for each combination of the possible values of the variables. If the ratio is above a certain threshold all the members of that groups get the value 1 for the new dimension, otherwise 0. Accuracy of the model is done by using cross validation and permutation tests, in simpler words it means that it reshuffles the data randomly and recalculates the model many times to get an estimate of the models certainty. Because of that MDR can be slow. However it is still usually faster than exhaustive search with regression methods.[1, 59]

MDR can been used for gene-environment interaction but requires modifications since MDR can only handle binary variables. There are extensions that can use continues variables, however these are regression based so these will be slower than regular MDR.[2]

A simple example of MDR using exclusive or (XOR). XOR is an logical operator that is true if one and only one of its two variables is true. We have 4 possible combinations and an occurrence for each of them, see table 2.5. The combination (1,0) and (0,1) both have one case with outcome 1 so MDR will classify them as 1 in the new variable Z, the other two combinations have outcome 0 so will be classified with Z=0, see table 2.6. From here it is easy to make an predictor from Z to the outcome Y by comparing the values.

Y	$\mathbf{X_1}$	$\mathbf{X_2}$
1	1	0
1	0	1
0	0	0
0	1	1

Table 2.5: *XOR table with outcome* **Y** *and variables* $\mathbf{X_1}$ *and* $\mathbf{X_2}$.

Y	**Z**
1	1
1	1
0	0
0	0

Table 2.6: *XOR table with* $\mathbf{X_1}$ *and* $\mathbf{X_2}$ *combined into* **Z** *using MDR.*

2.6.2.2 Random Forest

RF is an ensemble learning method[64]. Ensemble methods combine multiple models to improve performance. RF takes randomized samples of the data and builds decision trees on each of them. These trees are then combined to form the classifier. Usually hundreds or thousands of trees are used depending on the problem[64]. One of the most popular variants of Random Forest for GWAS is Random Jungle[65].

It has been shown in high dimensional data that RF tends to only rank interacting factors high if they have strong marginal effects[66]. Also the ranking of the variables does not indicate which factor it is interacting with either since it is based on the joint distributions[2]. How to incorporate the environmental factors in RF is also not obvious and using variables with very different scales can bias the results[2].

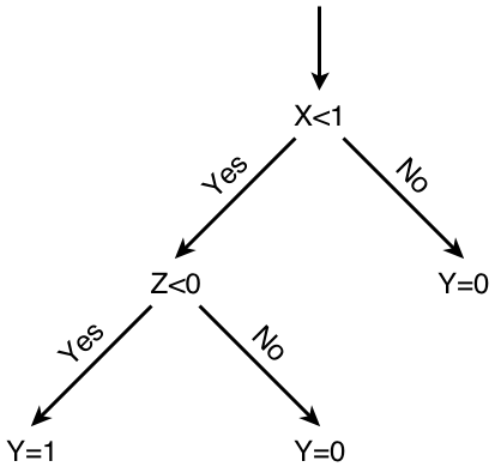


Figure 2.19: *A example of a decision tree*

Chapter 3

Implementation

This section will explain the implementation of the program starting with a short summary of how JEIRA and GEISA works. After that is a more thorough look at the structure and implementation of CuEira.

The input files for all these programs are PLINK data files. However CuEira can only read the PLINK files in binary format. They also read a separate tab delimited file that contains the environmental data and covariates if any. For CuEira the covariates and environmental data is split in two files.

3.1 Terminology

An enum is a data type consisting of a set of named values. This is useful when the variable has a distinct set of values that makes more sense as text rather than something else. For instance gender can be stored as enum with the two values MALE and FEMALE, the enum values are commonly write in capital letters. The gender could be stored an int, however it is easier to understand the meaning of MALE than just a number and it also avoids potential errors with incorrect values. An enum can only have one of it's possible values while an int is not restricted in the same way.

3.2 JEIRA and GEISA

The descriptions and structures in this section applies to both JEIRA and GEISA since their underlying structure is the same, however the focus is on GEISA. They are written in Java and uses Javas built-in functions for concurrency.

GEISA has three dependencies, two are provided as JAR files in the distribution so the user doesn't need to get them. Maven is only needed if the user wants to compile the program from the source.

Apache Commons CLI
: Provides command line interface, e.g. it parses the options when using the program.

Apache Commons Math
Provides matrix and vector classes, linear algebra, etc.

Apache Maven
Used to compile Java programs.

The implementation uses the producer consumer pattern described in section 2.4.2. The main thread creates a queue of tasks. All the result producers iterates over this shared queue and outputs results. These results are placed in a queue a consumed by another thread that handles the results. The program reads all data at the start which means it can get memory problems for large datasets. JEIRA and GEISA are both largely without tests. However there are a few unit tests for some of the basic storage classes and the binary file reader.

3.3 CuEira

This section will explain the structure of CuEira and how it works starting with a general overview. Later the parts are explained and finally how the parts are combined to form the program. C++ was chosen because it is generally fast and have good memory handling.
c++ has no garbage collector, speed but need to track self good because can delete old snp data
cuda vs opencl more interested in cuda opencl is a bit messy with the code modules makes it easier to swap if wanted.
downside of not Java more plattform dependent less good mocking and testing, junit and mockito are good. google test is good, google mock too but not super easy. due to how c++ works
The basis of the program is that each SNP is handled independently and a SNPs data is only read from the file when its needed. This reduces memory usage since only a few SNPs are stored in the memory at the same time. A number of worker threads share a queue of the SNPs and with each worker thread corresponding to a stream on a GPU. The program is split into two stages, initialisation and calculations. The initialisation part is done by the main thread and reads all the information except the SNP data. In the calculation step each worker threads fetches a SNP from the queue, reads its data, calculates the model, writes the results and then starts again with the next SNP until the queue is empty.

Copying of the classes is avoided to save performance. Most of the classes have the copy and move constructor and operator deleted to prevent mistakes since the default copy and move is naive and therefore usually flawed when dealing with classes with pointers. Since they are not needed allowing the default ones to be there will likely cause errors if they are used by accident.

Factories are used at several places to help satisfy the dependency inversion principle. Without the factories some objects can not be mocked and therefore a proper unit test can not be performed.

To improve the codes clarity several things were done. There are several namespaces to separate different parts of the program and to reduce clutter. For the same reason all classes have appropriate names and abbreviations were avoided in most cases. Comments were used sparsely because the names should be clear enough and explain what the code does that way. Comments are also easy to forget to update when the code is changed.

There are timers at various places that can be used to profile parts of the program. Some of them count how much time is spent waiting at locks or specific parts of the calculations.

3.3.1 Dependency and Compiling

CuEira have some dependencies. The dependency on MKL and Intels compiler is something that can be changed. BLAS shares the interface so another BLAS library could work with updates to the CMake file.

Make
Used for compiling

CMake
Used to create Make files

Boost
Boost is a collections of C++ libraries. For CuEira it is used for file handling, string operations and parsing the command line options among others.

CUDA
NVIDIAs library for GPU, explained in section 2.3

CUBLAS
BLAS for CUDA

MKL
Intels BLAS library

Intel Compiler
Intels C/C++ compiler

Google Test and Google Mock
Unit test and mocking framework. They are provided in the CuEira source so the user does not need to install them separably.

CuEira is compiled by using *CMake* and *Make*. CMake generates Make files from a CMakeLists file. Make files sets rules for how the program should be compiled. Compiling can be complicated and using CMake is one way to make it easier. One of the main advantages of CMake is the modules and the command *find package* which can find libraries by searching for them using common paths. They also help with linking the library with the program and can sometimes also provided other functions. For instance the find package for CUDA has commands to compile CUDA code. There is also support for cross platform compiling which is something that could be used in the future to compile CuEira for other operative systems than Linux.

The main CMake file needs to know what and where all the source files are. This is done by using nested CMake files, each directory has an CMakeLists file that adds it's sub directories and sets all the source files to the previous level. There are four types of lists of source files each is compiled into a library and then linked together. There is one list for all the tests, one for the files that needs to be compiled with nvcc, one for other GPU related code and then one with the host code.

The CMake file contains several options that controls how CuEira is compiled. For instance one of them sets if it should be single or double precision. More details on how to compile and use CuEira can be found in appendix E.

3.3.2 Storage

A set of container classes holds the data. There are two types of them, the basic matrix and vector classes and data containers. The data containers are responsible for handling the data for the model and performing the recoding. They use the vector classes for the actual storage.

There is three types of matrix and vector classes which are all stored in column major format. The set with the Device prefix is for storage on the device and the other two are for storage on the host and shares a common interface. The difference between the host sets is that the ones with Pinned prefix uses pinned memory while Regular does not.

Other basic information such as the information about individuals, environment factors and SNPs are stored in simple classes where each individual, environmental factor or SNP is a single object. *Enums* are used to store some of them, enums are a data type previously explained in section 3.1.

The SNPs are stored in the class **DataQueue**. **DataQueue** provides the function **next** that fetches a SNP from the queue and returns it. It has a lock so it can be shared among all worker threads.

The **EnvironmentFactorHandler** is responsible for keeping track of the **EnvironmentFactors** and it's corresponding data. When constructed it updates the EnvironmentFactors variable type. This is done because it can sometimes be needed to know if a variable is binary or not. It also provides functions to access the **EnvironmentFactors** and the corresponding vectors.

There is three data container classes, **SNPVector**, **EnvironmentVector** and **InteractionVector**. They all have the responsibility to keep track of the recoded data and to perform the recoding when their **recode** function is used. The **recode** function takes an **Recode** enum which tells it which type of recoding it should perform. The **SNPVector** also holds the original SNP data. **EnvironmentVector** doesn't need to since **EnvironmentFactorHandler** has that responsibility and **InteractionVector** does not have any kind of original data.

The **DataHandler** is responsible for keeping track of the current combination of data containers being used and to iterate to the next one when its **next** function is called. When the **next** function is used a couple of things happen. If all environment factors have been calculated together with the current SNP it deletes the SNPs information, asks the queue for a new one and reads its data and switches to the first environment factor. Otherwise it moves on to the next environment factor. It then updates the InteractionVector and the Statistics classes to create the contingency table and the allele frequencies. After that the next set of variables is ready to be used.

3.3.3 File Input and Output

CuEira reads five files which are in four different formats. There are five classes for reading the files, one for each file.

Three of the files are PLINK files in binary format. The fam file has the information about the individuals, the bim file has the names of the SNPs and their alleles. The genetic information for each individual is in the binary bed file. More details on PLINK files can be found in appendix B.1. The classes used to read them are **BimReader**, **BedReader** and **FamReader**.

The other two files are CSV files. One of them contains the covariates and the other the environmental factors. **CSVReader** is responsible for reading them and storing the data in a matrix. The fifth reader class is **EnvironmentCSVReader** which inherits **CSVReader**. Its purpose is to convert the matrix read by **CSVReader** with the environment data to **EnvironmentFactorHandler**.

All the information except the genetic data in the bed file is read during the programs initialisation. When a new SNP is being used for calculations its data is read from the bed file. Most of the memory needed to store the data is in the bed file so only reading it when needed save a lot of memory.

There is only one writer, the **ResultWriter**, and as its name suggests its responsibility is to write the results. The results are written immediately and then discarded to save memory. There is an lock on the writing so several threads can share it. The results will therefore be written in the order they are completed. The format of the output file is described in appendix B.

3.3.4 Initialisation of the Variables, Recoding and Statistic Model

The initialisation of the **SNPVector** and **EnvironmentVector** is done by calling the recoding with ALL_RISK from the **Recode** enum. The **recode** function calls different functions based on the enum. If it is ALL_RISK it calls a function that copies the data from the vector holding the original data to vector that holds the data that will be used for the model. **InteractionVector** does recoding and initialisation the same way because in both cases the interaction data is the multiplication of the elements of the **SNPVector** and **EnvironmentVector**. The algorithms for the recode functions are shown in algorithm 6, 4 and 5.

For the additive statistical model all the elements in **SNPVector** and **EnvironmentVector** needs to be set to 0 when the corresponding element in **InteractionVector** is 1. Both classes have an function named **applyStatisticModel** that performs it. It takes the **InteractionVector** as an parameter and the algorithm is the same for both **SNPVector** and **EnvironmentVector**. Its is shown in algorithm 7.

DataHandler has the overall responsibility for recoding and applying the statistical model, it calls the data containers functions as needed. It also updates the **ContingencyTable** after recoding since the distribution of the groups changes when recoding.

Algorithm 4: SNPVector protective recoding

Data: Asdf

Algorithm 5: EnvironmentVector protective recoding

Data: Two vectors, $originalData$ and $recodedData$

```
for (int i = 0; i < numberOfIndividualsToInclude; ++i)
{
    if ((*originalData)(i) == 0)
    {
        (*recodedData)(i) = 1;
    }
    else
    {
        (*recodedData)(i) = 0;
    }
}
```

Algorithm 6: InteractionVector recoding

Data: $EnvironmentVector$ and $SNPVector$

```
const HostVector& envData = environmentVector.getRecodedData();
const HostVector& snpData = snpVector.getRecodedData();

for (int i = 0; i < numberOfIndividualsToInclude; ++i)
{
    (*interactionVector)(i) = envData(i) * snpData(i);
}
```

Algorithm 7: Applying statistic model

Data: $recodedData$, vector with the variable
$interactionVector$, vector with the interaction variable

```
for (int i = 0; i < numberOfIndividualsToInclude; ++i)
{
    if (interactionVector(i) != 0)
    {
        (*recodedData)(i) = 0;
    }
}
```

3.3.5 Wrappers

CUDA and BLAS functions and interfaces are in C style. To make it easier to use and replace if needed they were put in wrappers. These wrappers are in C++ style and do the actual function calls to CUDA and BLAS. Since CUDA and BLAS contains a lot of various functions functions were added to the wrappers when needed.

There are several reasons to use wrappers. One is that some of the functions does not use exceptions but returns an error code instead. This forces the program to check these error codes. This is hidden by using a wrapper that can throw the correct exception based on the error code. Exceptions are also better for performance because it removes the need for if statements checking the returned error codes. The compiler can optimize better with checks for exceptions(e.g. try catch statements) because it can assume that the occurrence of exceptions is low. However usually checks for exceptions are not needed because they commonly signify an error that the program can not recover from.

Another reason is the lack of object orientation. For instance the BLAS interface standard is not object oriented which causes its functions to have a lot of parameters. Matrix vector multiplication with MKL(Intels version of BLAS) is shown in 5. The function call uses 12 parameters, most of which are of similar types. One time during the thesis it took a day to find an error that was in one of the CUBLAS calls. Due to a parameter being 1 instead of 0 it performed the operation $y = z*x+y$ instead of $y = z*x$. These types of errors are much easier to make when a function has a larger number of similar parameters. MKL is wrapped in **MKLWrapper**. Its wrapped version of matrix vector multiplication, shown in 6, has 5 parameters of which two have default values.

As is sometimes common BLAS function names are heavily abbreviated. For instance a function for matrix vector multiplication is **sgemv**, the *s* stands for single precision, *ge* for general, *m* for matrix and *v* for vector. General means that it is standard matrices and vectors, e.g. not triangular or sparse. The equivalent function in **MKLWrapper** can be found in example 6. The name matrixVectorMultiply instantly tells the user what it does. As mentioned earlier in section 3.3 the CuEira is either single or double precision depending on an option when compiling so **MKLWrapper** doesn't mention the precision. All matrices and vectors are also general. However even if the function name would contain that information too it would still not be so long that it was in the way. A possible name by using namespaces for the precision and type could be General::SinglePrecision::matrixVectorMultiply.

CUBLAS is wrapped together with the kernels made for CuEira in **KernelWrapper** in a similar way to how MKL is wrapped in **MKLWrapper**. **KernelWrapper** is explained in section 3.3.6. Other CUBLAS utility and CUDA functions are wrapped in **CudaAdapter**.

The CUDA streams are managed by a **Stream** class and its factory. When the **StreamFactory** creates a new Stream object it creates a new CUDA stream, a new CUBLAS handle and associates the CUBLAS handle with the new stream.

Example 5: Single precision matrix vector multiplication using MKL, $result = \alpha * matrix * vector + \beta * result$

Data: *layout*, storage type of the matrix, either column major(CblasColMajor) or row major(CblasRowMajor)
trans, use transpose, conjugate or neither on the matrix
matrix, *vector* and *result* are float pointers
α and β are constants
m, number of rows in the matrix
n, number of columns in the matrix
ld_matrix, leading dimension of the matrix
inc_vector and *inc_result* are the increments of the elements

cblas_sgemv(*layout*, *trans*, m, n, α, *matrix*, *ld_matrix*, *vector*, *inc_vector*, β, *result*, *inc_result*);

Example 6: Matrix vector multiplication using MKLWrapper, $result = \alpha * matrix * vector + \beta * result$

Data: *matrix*, HostMatrix
vector, HostVector
result, HostVector
α, constant, default 1
β, constant, default 0

matrixVectorMultiply(*matrix*, *vector*, *result*, α, β);

The **Stream** class has functions to retrieve them and if the Stream object is destroyed it destroys both the CUDA stream and the CUBLAS handle. Each Stream object is associated with a **Device** object. The **Device** class represents a device(i.e. GPU) and since a thread can only issue commands to one device at a time the **Device** class has functions to set it as the active one and check if itself is the device active or not. The outcomes of the individuals does not change so it can be shared between all streams on the device and the **Device** class is responsible for the vector with the outcomes on the device.

The transfers to and from the device are handle by two classes, **DeviceToHost** and **HostToDevice**. They perform asynchronous transfers on the corresponding matrix and vector class. It can either create a new container to transfer to or transfer to an given place in the memory. The former is useful for instance when transferring different vectors to into a combined matrix.

3.3.6 Kernels

All the kernels are wrapped together with the CUBLAS kernels in **KernelWrapper**. All the kernels made for CuEira are of the same type, they all do operations on each element of vectors. This makes their structure simple, each thread does the operations on one index. An example of one of the kernels and

its wrapper function can be found in algorithm 8 and algorithm 9.

Algorithm 8: Kernel for vector addition

Data: *vector*1, *vector*2 and *result* are vectors as pointers,
length is the length of the vectors

```
__global__ void ElementWiseAddition(const PRECISION* vector1,
const PRECISION* vector2, PRECISION* result, const int length)
{
    int threadId = blockDim.x * blockIdx.x + threadIdx.x;

    if (threadId < length)
    {
        result[threadId] = vector1[threadId] + vector2[threadId];
    }
}
```

Algorithm 9: Wrapper for the kernel in algorithm 8

Data: *vector*1, *vector*2 and *result* are DeviceVectors of same length
cudaStream is the stream for the kernel
numberOfThreadsPerBlock is 256

```
void elementWiseAddition(const DeviceVector& vector1, const
DeviceVector& vector2, DeviceVector& result)
{
    //Error check for the length of the vectors in the source code removed

    const int numberOfBlocks = std::ceil(((double)
    vector1.getNumberOfRows()) / numberOfThreadsPerBlock);

    Kernel::ElementWiseAddition<<<<
    numberOfBlocks, numberOfThreadsPerBlock, 0, cudaStream >>>>

    (vector1.getMemoryPointer(), vector2.getMemoryPointer(),
    result.getMemoryPointer(), vector1.getNumberOfRows());
}
```

3.3.7 Model

The calculations are done by using a group of **Model** classes. A list of these classes and their responsibilities is shown below.

Model
The model to calculate.

ModelConfiguration
Holds the data for the Model by using the builder pattern.

ModelResult
Contains the results from one Model on one set of data.

CombinedResults
The results for one set can consist of more than one ModelResult. CombinedResults have that responsibility.

ModelHandler
Has a **next** function to iterate over the data delegating it to DataHandler. It also has a pure virutal function **calculate** which should be used for calculating the Models and collecting the results.

ModelInformation
Contains the ContingencyTable and AlleleStatistics.

The point of these classes is that they provided common interfaces and are easy to extend. This makes it easy to later add or change which models are calculated and how their results are handled. The next section will explain how the Model base classes were extended to form the classes for the LR model.

3.3.7.1 Logistic Regression

The LR model is implemented using the **Model** base classes talked about in the previous section. There is a CPU version and a GPU version using CUDA. Some classes are shared between them because of common elements. The structure is shown in figure 3.1. A more detailed layout of the **ModelHandler** for **LogisticRegression** is shown in figure 3.2.

The LR algorithm is split into several protected functions so that each part can be tested individually. The parts are then used in the **calculate** function. For **CudaLogisticRegression** the parts corresponding to line 10 and 11 in algorithm 3 are done on the host. All the other parts of the LR algorithm are done by using CUBLAS or the kernels explained in section 3.3.6. The calculations for line 10 and 11 are in the **LogisticRegression** class so the code is shared between **CpuLogisticRegression** and **CudaLogisticRegression**.

Line 10 and 11 are done on the host instead of the device because the inverse of the information matrix(variable **J** in the algorithm) is done by SVD and no suitable GPU function for it was found. The reason for using SVD instead of normal matrix inverse was previously explained in section 2.6.1. CUBLAS does not provide an SVD function. However there are other libraries that do, CULA Tools[47] and MAGMA[48], however neither could be used.

CULA Tools could not be used because it uses streams internally in the functions so it can not be executed on streams[67]. It also uses device synchronisation which means that the all streams will be synchronised every time the function is used[67]. MAGMA can use streams for its BLAS functions, however it does not for the other functions including the SVD[68].

The size of the vectors and matrices used in that part are also small, their size is 4 + the number of covariates. However it can sometimes be better to

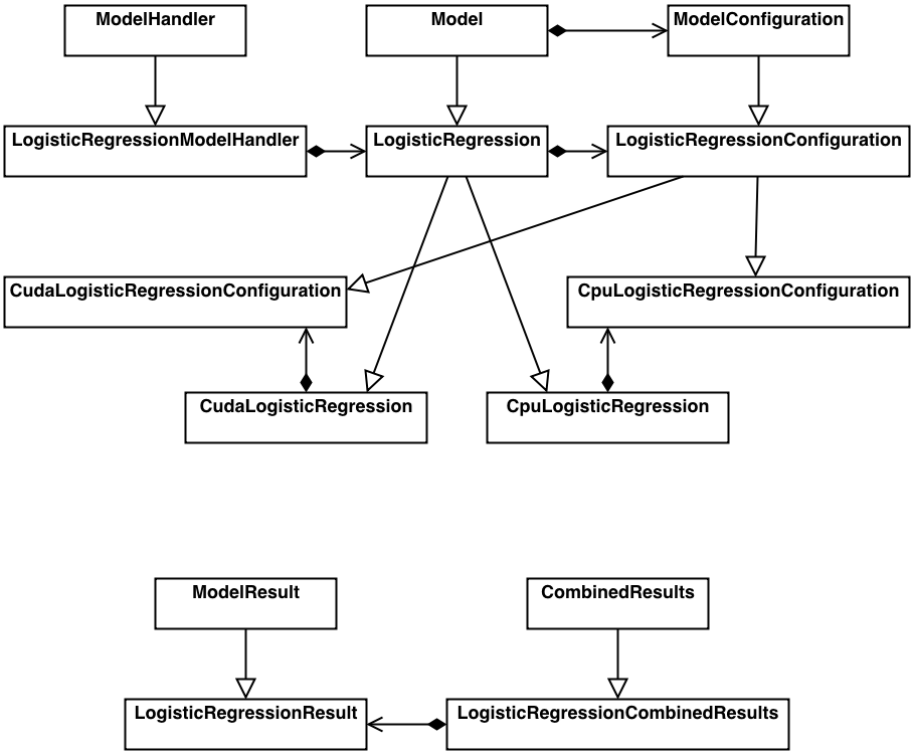


Figure 3.1: *Overview of the logistic regression classes. White arrow is inheritance. Diamond is that the class with the diamond has instances of the other class.*

calculate even small matrices and vectors on the device to avoid the transfers that would be needed otherwise[34]. An suitable GPU SVD kernel might be faster.

After transfers it is necessary to wait for it to finish to prevent errors with using the data before it has been transferred completely. This is done by syncing the thread with the stream, it forces the thread to wait until all current kernels and transfers on the stream is complete. However due to the problem with asynchronous transfers and older architectures described in section 2.3.2 there is also an option when compiling to synchronise after each kernel.

3.3.8 Worker Thread

A number of worker threads use the previously explained classes to perform the work. One worker thread corresponds to one stream on a GPU. Normally three streams are used per GPU but it can easily be changed by changing a parameter in the main **Configuration** class. The layout of the worker thread is shown in figure 3.3.

The thread has its own **DataHandler**, **ModelHandler**, **ModelConfiguration** and Model. Some objects are shared with the other workers, most notably the **ResultFileWriter** and the **DataQueue**. Each thread loops over

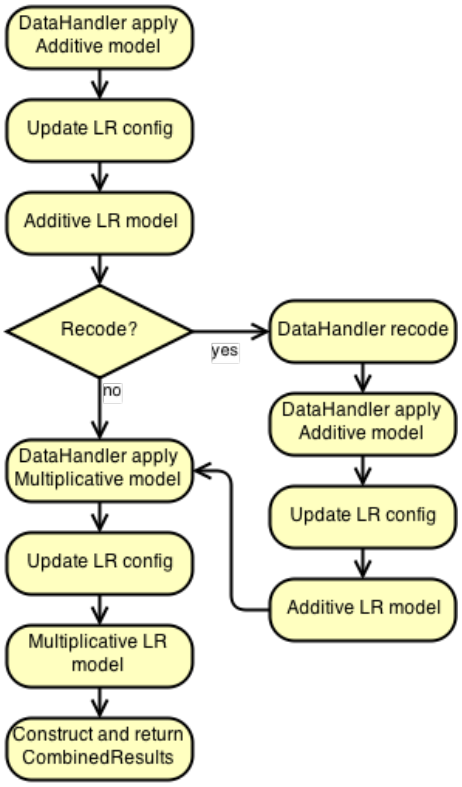


Figure 3.2: *Layout of the LogisticRegressionModelHandler*

the **ModelHandlers next** function, tells the **ModelHandler** to calculate and then sends the results to the **ResultFileWriter**.

3.3.9 Testing

Almost all classes have a corresponding unit test that tests its functionality in isolation. However the matrix and vector classes are never mocked out because they are too basic and fundamental. It does mean that if there are multiple errors when testing and the tests for the matrix and vectors failed the fault likely is with those. However the number of integration tests are few and there is a need to create more.

Since mocks are created by inheritance from the class to be mocked all functions need to be virtual. Some classes also have empty protected constructors to be used by the mock, others have helper functions to construct the mock. This goes against some parts of clean code but is necessary because of how C++ and mocking works. It could potentially be improved by using private constructors and friends to give the mock access.

The data for the tests where calculated by hand or by using Matlab.

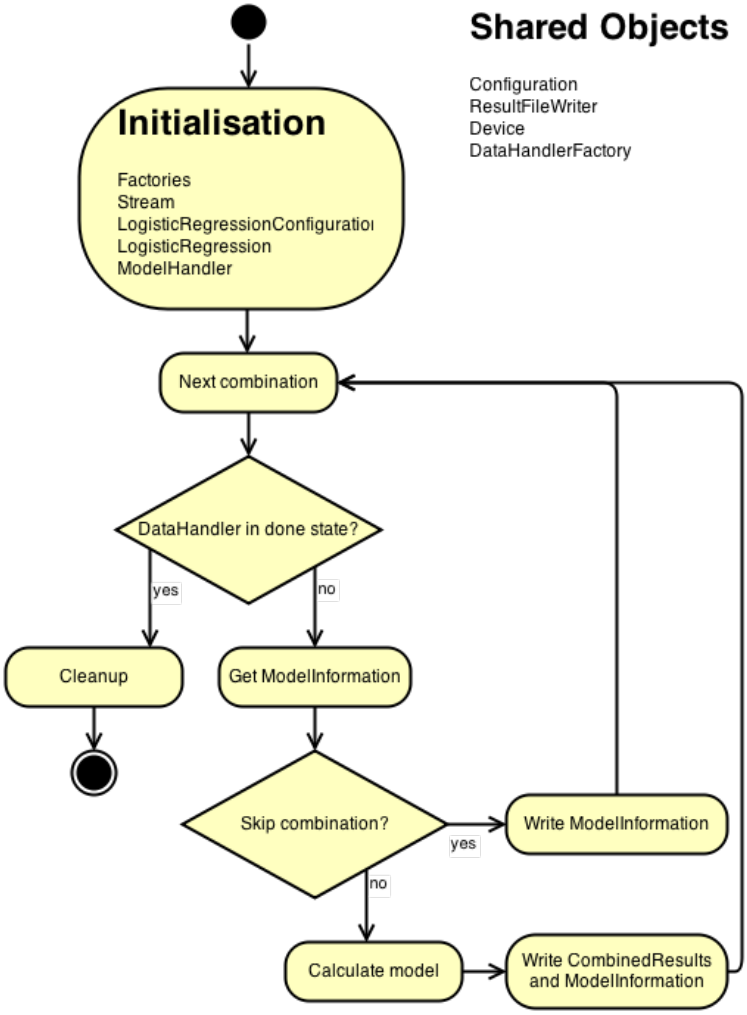


Figure 3.3: *Structure of the worker thread*

3.3.10 Memory Usage

The memory usage is low because only the SNPs whose models currently is being calculated is stored in memory. This also means that most of the memory usage depends on the number of individuals not the number of SNPs especially since the SNP class is small. The memory is also bound by the GPU rather than host memory because of the relative size and that the GPU and host need about the same about of memory. However it does depend on the number of GPUs. A Tesla GPU has 6GB of memory[33], how many individuals can it fit at most?

Float=4 bytes, double=8 With n number of individuals and m, $m = 4+$ number of covariates, one set of memory for the LR model needs:
The outcomes, size n, shared for all streams

Predictors, n*m Probabilities n Score m Beta m Work area n Work area n*m Information matrix m*m
With three streams the number of floats needed is

$$6GB = (n + 3(2 * n * m + 2 * n + 2 * m + m^2)) * precsion \tag{3.1}$$

Breaking out n

$$n = \frac{\frac{6GB}{6m+7} - m^2 - 6m}{precision} \tag{3.2}$$

See table 3.1. Even with 100 covariates and double precision it can store 1.2 million individuals.

Number of covariates	Single precision	Double precision
0	48.4	24.2
5	24.6	12.3
10	16.5	8.2
20	9.0	5.0
100	2.4	1.2

Table 3.1: *The number of individuals in millions needed to fill 6GB with the given precision, number of covariates and three streams.*

n individuals s snps e env c cov
shared s snps e environment factors n persons
per thread
lr
snp string vector int max längd=4 string string int int bool
envfactor string int
person string int int bool
The strings are short but there is significant overhead probably.
alt form:

Chapter 4

Results

Intro

To make the comparison of the performance of the programs more fair a couple of details in CuEira was changed. With the changes CuEira assigns the risk allele the same way as the older programs and doesn't use recoding on the multiplicative model. The MAF threshold was also set to zero. The way the risk allele is calculated in JEIRA and GEISA is not according to the definition earlier in section 1.3.1, the way JEIRA and GEISA does it is shown in appendix C.

4.1 Data

Acpapos, missing data problem
Simulated data, random, varying sizes python script

4.2 Hardware Specifications

Cluster Zorn at PDC was used[69].
Zorn specs
8 nodes 92 gb ram 2 Intel Xeon E5620, 2.4 ghz, 4 cores 3 tesla m2090
Due to errors only one 1 GPU can be accessed. Because of this the login node had to be used for testing with multiple GPUs. It's specs are
4 tesla c2050 50 gb ram?
farliy old gpus

4.3 CuEira vs GEISA

Bla bla bla
200k individuals extrapolate for GEISA. It was cut after about a hour and 1040 SNPs calculated. With shown in figure 4.1, without shown in figure 4.2
speedup 1 gpu normal node

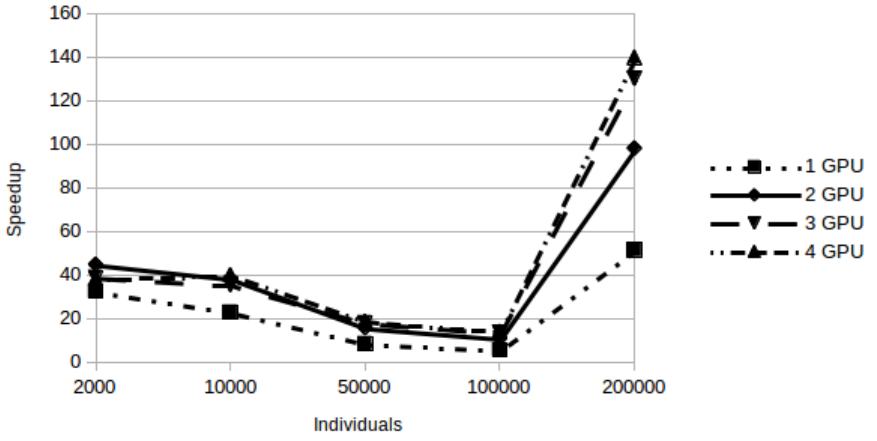


Figure 4.1: *Speedup GEISA vs CuEira. GEIRA 200 000 individuals extrapolated. 10 000 SNPs.*

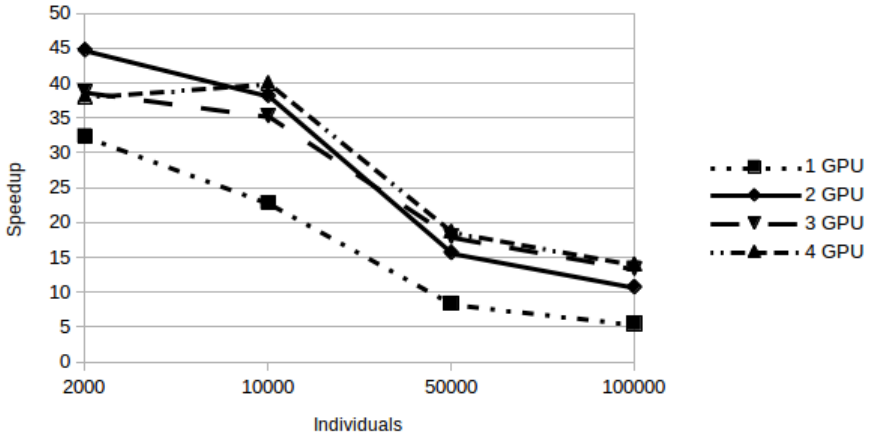


Figure 4.2: *Speedup GEISA vs CuEira. Without 200 000 individuals data. 10 000 SNPs.*

4.4 More detailed CuEira stuff

speedup for gpu
efficency

4.5 Number of streams

vary number of streams

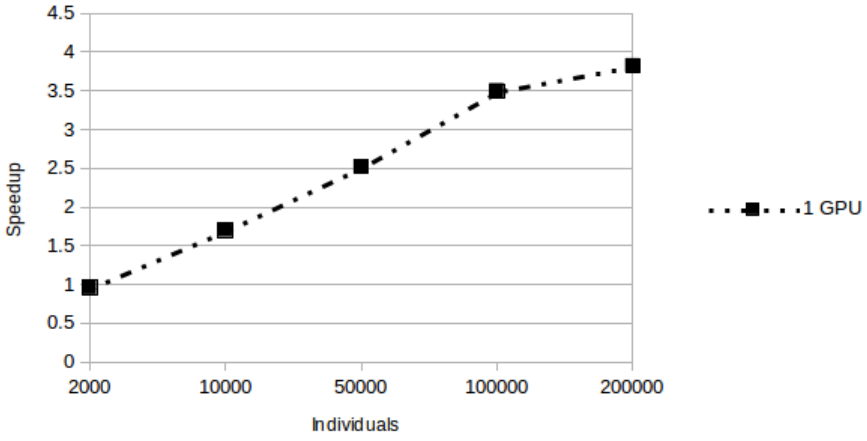


Figure 4.3: *Speedup GEISA vs CuEira. 10 000 SNPs.*

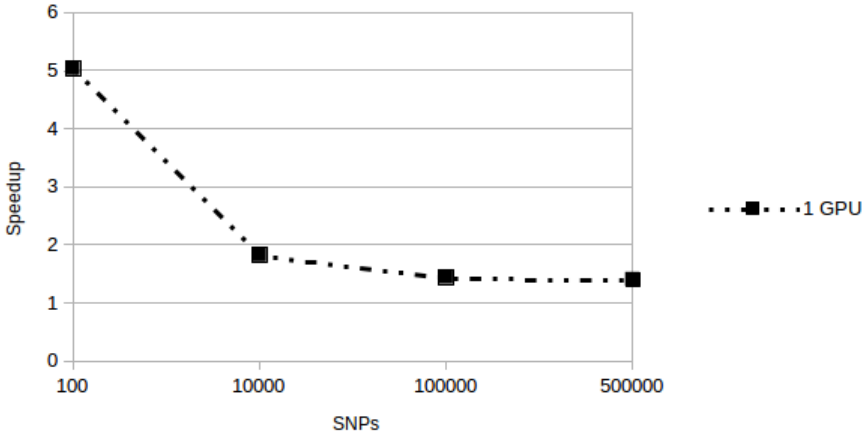


Figure 4.4: *Speedup GEISA vs CuEira. 10 000 Individuals.*

4.6 Single vs Double Precision

no difference in the results. time increased

4.7 Syncing

minor increase with syncing

4.8 Profiling

personhandler, longer at random?

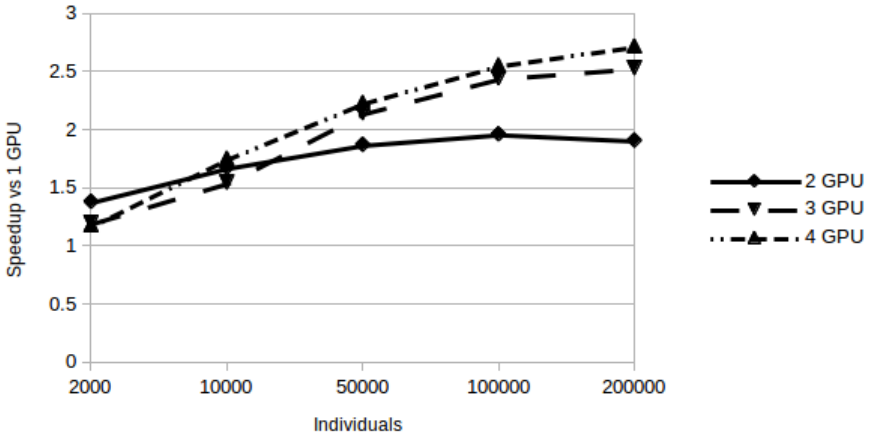


Figure 4.5: *Speedup vs 1 GPU on login node. 10 000 SNPs.*

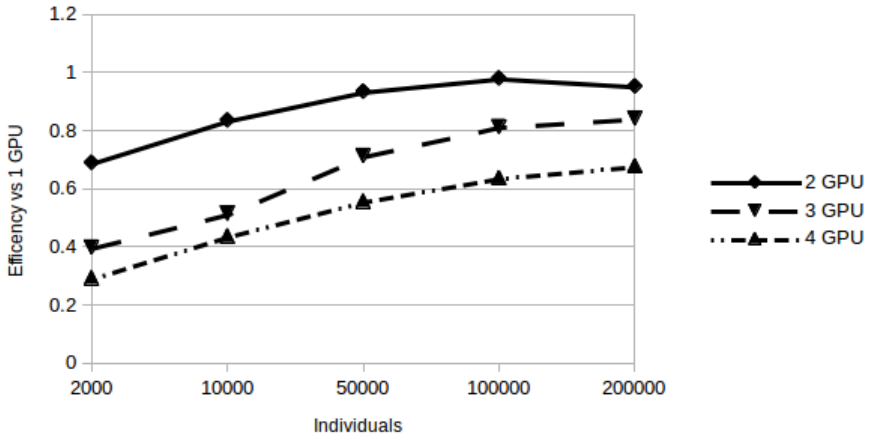


Figure 4.6: *Efficiency vs 1 GPU on login node. 10 000 SNPs.*

time spent at various parts 1/2 gpu vs 4 gpu, cpu vs gpu on LR
nsight, 100 snps, 2 gpu vs 4 gpu sync/vs no sync

4.8.1 Locks

Very little time spent

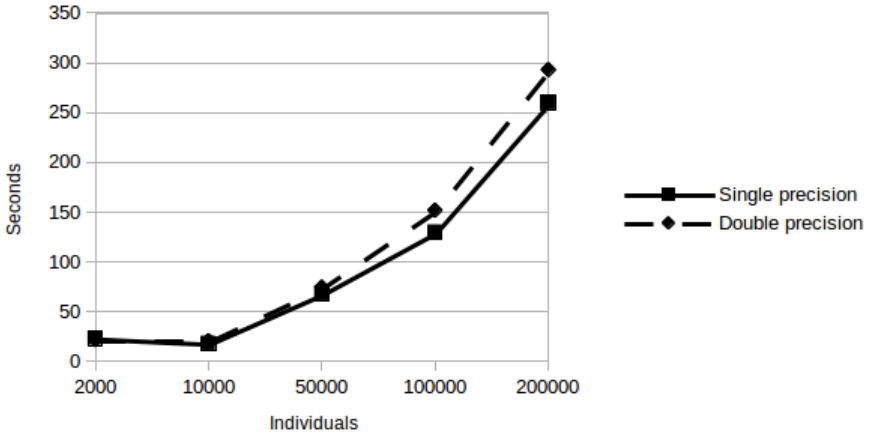


Figure 4.7: *Execution time with 4 GPUs with either single or double precision. 10 000 SNPs.*

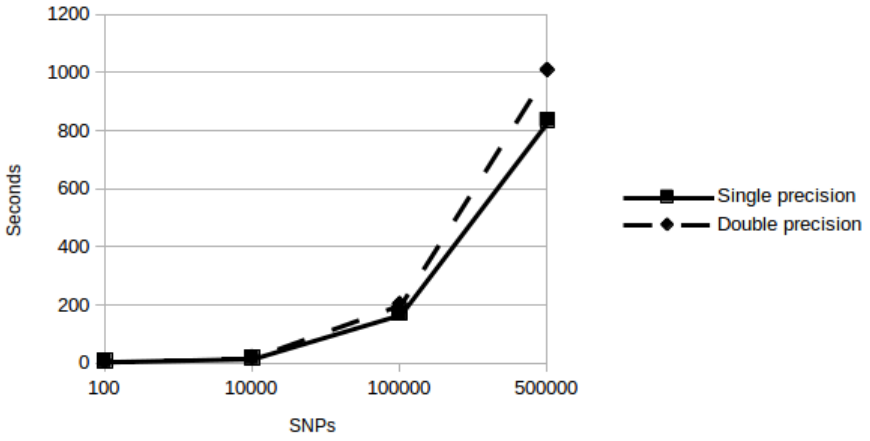


Figure 4.8: *Execution time with 4 GPUs with either single or double precision. 10 000 individuals.*

4.9 Compiler Optimizations

4.10 Other

mkl sequential

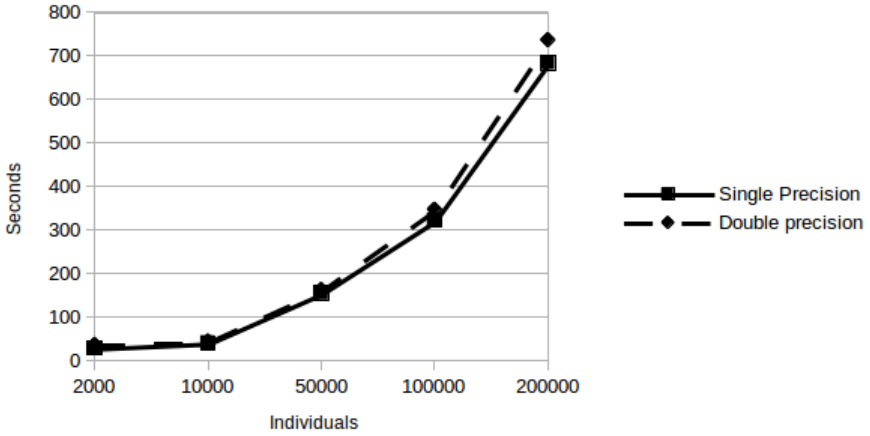


Figure 4.9: *Execution time with 1 GPU on login node with either single or double precision. 10 000 SNPs.*

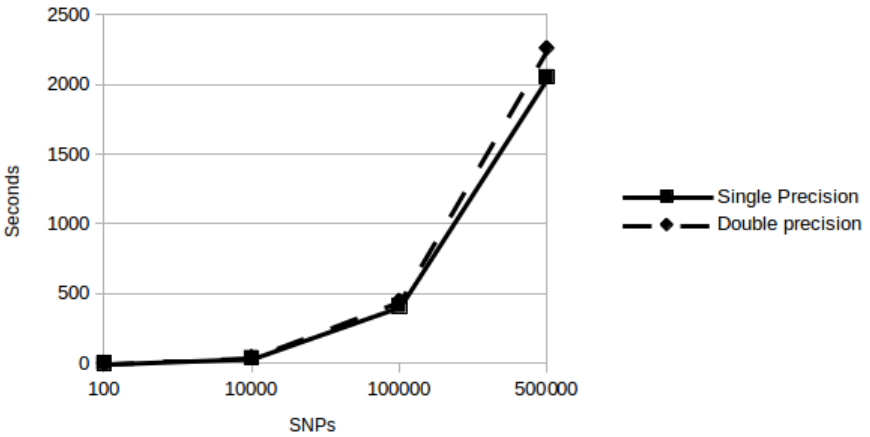


Figure 4.10: *Execution time with 1 GPU on login node with either single or double precision. 10 000 individuals.*

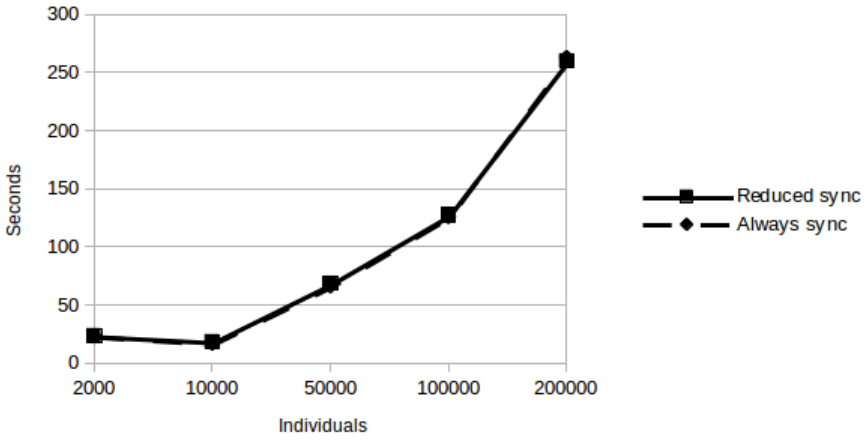


Figure 4.11: *Execution time with 4 GPUs with either synchronisation or not. 10 000 SNPs.*

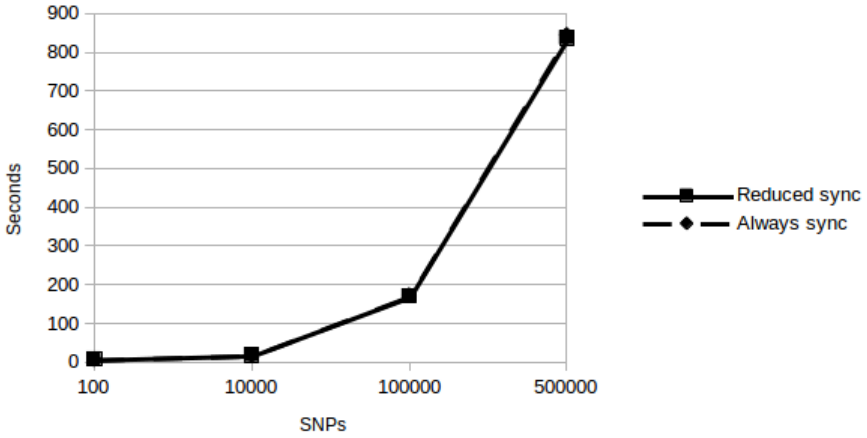


Figure 4.12: *Execution time with 4 GPUs with either synchronisation or not. 10 000 individuals.*

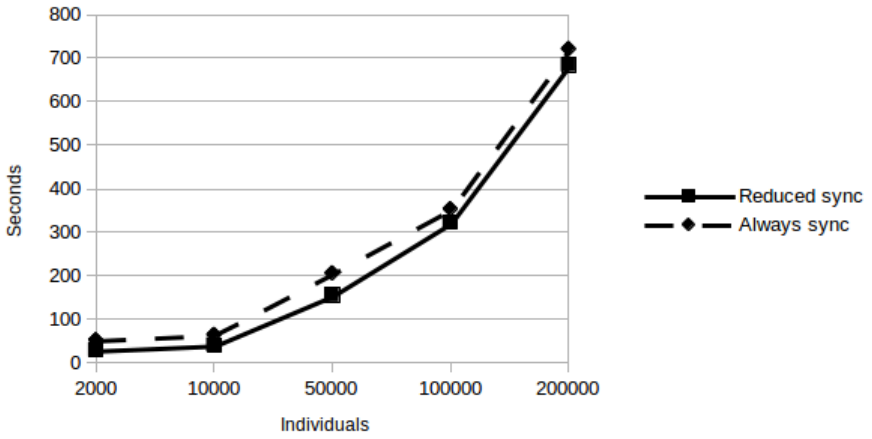


Figure 4.13: *Execution time with 1 GPU on cluster node with either synchronisation or not. 10 000 SNPs.*

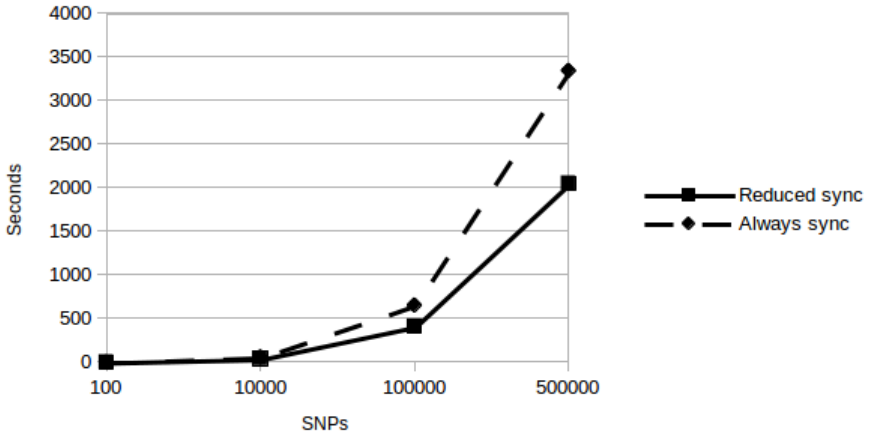


Figure 4.14: *Execution time with 1 GPU on cluster node with either synchronisation or not. 10 000 individuals.*

Chapter 5

Discussion and Conclusions

Chapter 6

Outlook

Bibliography

[1] H. J. Cordell, "Detecting gene–gene interactions that underlie human diseases," *Nature Reviews Genetics*, vol. 10, no. 6, pp. 392–404, 2009.

[2] S. J. Winham and J. M. Biernacka, "Gene–environment interactions in genome-wide association studies: current approaches and new directions," *Journal of Child Psychology and Psychiatry*, vol. 54, no. 10, pp. 1120–1134, 2013.

[3] B. Ding, H. Källberg, L. Klareskog, L. Padyukov, and L. Alfredsson, "Geira: gene-environment and gene-gene interaction research application," *European Journal of Epidemiology*, vol. 26, no. 7, pp. 557–561, 2011.

[4] H. Mahdi, B. A. Fisher, H. Källberg, D. Plant, V. Malmström, J. Rönnelid, P. Charles, B. Ding, L. Alfredsson, L. Padyukov, *et al.*, "Specific interaction between genotype, smoking and autoimmunity to citrullinated α-enolase in the etiology of rheumatoid arthritis," *Nature genetics*, vol. 41, no. 12, pp. 1319–1324, 2009.

[5] K. J. Rothman, S. Greenland, and T. L. Lash, *Modern epidemiology*. Lippincott Williams & Wilkins, 2008.

[6] C. Mann, "Observational research methods. research design ii: cohort, cross sectional, and case-control studies," *Emergency Medicine Journal*, vol. 20, no. 1, pp. 54–60, 2003.

[7] P. R. Burton, D. G. Clayton, L. R. Cardon, N. Craddock, P. Deloukas, A. Duncanson, D. P. Kwiatkowski, M. I. McCarthy, W. H. Ouwehand, N. J. Samani, *et al.*, "Genome-wide association study of 14,000 cases of seven common diseases and 3,000 shared controls," *Nature*, vol. 447, no. 7145, pp. 661–678, 2007.

[8] B. Goudey, D. Rawlinson, Q. Wang, F. Shi, H. Ferra, R. M. Campbell, L. Stern, M. T. Inouye, C. S. Ong, and A. Kowalczyk, "Gwis-model-free, fast and exhaustive search for epistatic interactions in case-control gwas," *BMC genomics*, vol. 14, no. Suppl 3, p. S10, 2013.

[9] G. Fang, M. Haznadar, W. Wang, H. Yu, M. Steinbach, T. R. Church, W. S. Oetting, B. Van Ness, and V. Kumar, "High-order snp combinations associated with complex diseases: efficient discovery, statistical power and functional interactions," *PloS one*, vol. 7, no. 4, p. e33531, 2012.

[10] S. Leem, H.-h. Jeong, J. Lee, K. Wee, and K.-A. Sohn, "Fast detection of high-order epistatic interactions in genome-wide association studies using information theoretic measure," *Computational Biology and Chemistry*, 2014.

[11] D. Sadava, D. Hillis, C. Heller, G. Orians, and W. Purves, *Life The Science of Biology*. Sinauer Associates, 8th ed., 2008.

[12] D. Uvehag, "Design and implementation of a computational platform and a parallelized interaction analysis for large scale genomics data in multiple sclerosis," 2013.

[13] D. Uvhage and H. Zazzi, "Geisa." `https://github.com/menzzana/geisa`, 2014.

[14] A. Ahlbom and L. Alfredsson, "Interaction: a word with two meanings creates confusion," *European journal of epidemiology*, vol. 20, no. 7, pp. 563–564, 2005.

[15] K. J. Rothman, *Epidemiology: an introduction*. Oxford University Press, 2002.

[16] A. Sjölander, W. Lee, H. Källberg, and Y. Pawitan, "Bounds on causal interactions for binary outcomes," *Biometrics*, 2014.

[17] A. Agresti, *Categorical data analysis*. John Wiley & Sons, Second ed., 2002.

[18] B. Lindgren, *Statistical theory*. CRC Press, fourth ed., 1993.

[19] H. T. O. Davies, I. K. Crombie, and M. Tavakoli, "When can odds ratios mislead?," *Bmj*, vol. 316, no. 7136, pp. 989–991, 1998.

[20] M. J. Knol, T. J. VanderWeele, R. H. Groenwold, O. H. Klungel, M. M. Rovers, and D. E. Grobbee, "Estimating measures of interaction on an additive scale for preventive exposures," *European journal of epidemiology*, vol. 26, no. 6, pp. 433–438, 2011.

[21] B. D. H. Foundation, "Smoking and oral health." `https://www.dentalhealth.org/tell-me-about/topic/sundry/smoking-and-oral-health`.

[22] J. M. Bland and D. G. Altman, "Multiple significance tests: the bonferroni method," *Bmj*, vol. 310, no. 6973, p. 170, 1995.

[23] G. Hager and G. Wellein, *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010.

[24] Intel, "Intel unleashes its first 8-core desktop processor." `http://newsroom.intel.com/community/intel_newsroom/blog/2014/08/29/intel-unleashes-its-first-8-core-desktop-processor`, 2014.

[25] C. Hoare, *Communicating sequential processes*. Prentice-Hall, Inc., 1985.

[26] W. F. Gilreath and P. A. Laplante, *Computer Architecture: A Minimalist Perspective: Dynamics and Sustainability*. Springer, 2003.

[27] U. Drepper, "What every programmer should know about memory," 2007.

[28] F. Abi-Chahla, "Intel core i7 (nehalem): Architecture by amd?." `http://www.tomshardware.com/reviews/Intel-i7-nehalem-cpu,2041-2.html`, 2008.

[29] Intel, "i7-5960x specification sheet." `http://ark.intel.com/products/82930`, 2014.

[30] NVIDIA, *NVIDIA CUDA C Programming Guide*, v5.5 ed., July 2013.

[31] J. Jeffers and J. Reinders, *Intel Xeon Phi coprocessor high performance programming*. Newnes, 2013.

[32] M. H. NVIDIA, "Maxwell: The Most Advanced CUDA GPU Ever Made." `http://devblogs.nvidia.com/parallelforall/maxwell-most-advanced-cuda-gpu-ever-made/`, 2014.

[33] NVIDIA, "Nvidia tesla-kepler product description." `http://www.nvidia.com/content/tesla/pdf/NVIDIA-Tesla-Kepler-Family-Datasheet.pdf`.

[34] NVIDIA, *NVIDIA CUDA C Best Practices Guide*, v5.5 ed., July 2013.

[35] Intel, "Intel Xeon Phi Coprocessor DataSheet, Document ID 328209 003EN." `http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-datasheet.html`, 2014.

[36] NVIDIA, *CUBLAS Library User Guide*, v5.5 ed., July 2013.

[37] L. S. Yung, C. Yang, X. Wan, and W. Yu, "Gboost: a gpu-based tool for detecting gene–gene interactions in genome–wide case control studies," *Bioinformatics*, vol. 27, no. 9, pp. 1309–1310, 2011.

[38] Z. Zhu, X. Tong, Z. Zhu, M. Liang, W. Cui, K. Su, M. D. Li, and J. Zhu, "Development of gmdr-gpu for gene-gene interaction analysis and its application to wtccc gwas data for type 2 diabetes," *PloS one*, vol. 8, no. 4, p. e61943, 2013.

[39] S. Lee, M.-S. Kwon, I.-S. Huh, and T. Park, "Cuda-lr: Cuda-accelerated logistic regression analysis tool for gene-gene interaction for genome-wide association study," in *Bioinformatics and Biomedicine Workshops (BIBMW), 2011 IEEE International Conference on*, pp. 691–695, IEEE, 2011.

[40] S. Chikkagoudar, K. Wang, and M. Li, "Genie: a software package for gene-gene interaction analysis in genetic association studies using multiple gpu or cpu cores," *BMC research notes*, vol. 4, no. 1, p. 158, 2011.

[41] J. Poznanovic, "Plinkgpu: A framework for gpu acceleration of whole genome data analysis," Master's thesis, School of Informatics, University of Edinburgh, 2010.

[42] R. Jiang, F. Zeng, W. Zhang, X. Wu, and Z. Yu, "Accelerating genome-wide association studies using cuda compatible graphics processing units," pp. 70–76, 2009.

[43] S. Almeida, "An introduction to high performance computing," *International Journal of Modern Physics A*, vol. 28, no. 22n23, p. 1340021, 2013.

[44] NVIDIA, "Kepler Tuning Guide." `http://docs.nvidia.com/cuda/kepler-tuning-guide`.

[45] Mark Harris, "How to Overlap Data Transfers in CUDA C/C++." `http://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/`.

[46] G. Ruetsch and B. Leback, "CUDA Fortran Asynchronous Data Transfers." `http://www.pgroup.com/lit/articles/insider/v3n1a4.htm`.

[47] "CULA Tools: GPU Accelerated Linear Algebra," 2010.

[48] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with gpu accelerators," in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pp. 1–8, IEEE, 2010.

[49] J. Hoberock and N. Bell, "Thrust: A parallel template library," 2010. Version 1.7.0.

[50] V. Volkov, "Unrolling parallel loops," *Tutorial at the*, p. 133, 2011.

[51] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2008.

[52] R. C. Martin, "Design principles and design patterns." `http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf`, 2000.

[53] G. Moraila, A. Shankaran, Z. Shi, and A. M. Warren, "Measuring reproducibility in computer systems research." `http://reproducibility.cs.arizona.edu/v1/tr.pdf`, 2013.

[54] J. Loeliger and M. McCullough, *Version Control with Git: Powerful tools and techniques for collaborative software development*. "O'Reilly Media, Inc.", 2012.

[55] X.-H. Sun and Y. Chen, "Reevaluating amdahlâĂŹs law in the multicore era," *Journal of Parallel and Distributed Computing*, vol. 70, no. 2, pp. 183–188, 2010.

[56] J. L. Gustafson, "Reevaluating amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[57] NVIDIA, "Nsight eclipse edition." `https://developer.nvidia.com/nsight-eclipse-edition`.

[58] A. Gyenesei, J. Moody, A. Laiho, C. A. Semple, C. S. Haley, and W.-H. Wei, "Biforce toolbox: powerful high-throughput computational analysis of gene–gene interactions in genome-wide association studies," *Nucleic acids research*, vol. 40, no. W1, pp. W628–W632, 2012.

[59] M. D. Ritchie, L. W. Hahn, N. Roodi, L. R. Bailey, W. D. Dupont, F. F. Parl, and J. H. Moore, "Multifactor-dimensionality reduction reveals high-order interactions among estrogen-metabolism genes in sporadic breast cancer," *The American Journal of Human Genetics*, vol. 69, no. 1, pp. 138–147, 2001.

[60] X. Wan, C. Yang, Q. Yang, H. Xue, X. Fan, N. L. Tang, and W. Yu, "Boost: A fast approach to detecting gene-gene interactions in genome-wide case-control studies," *The American Journal of Human Genetics*, vol. 87, no. 3, pp. 325–340, 2010.

[61] Y. Zhang and J. S. Liu, "Bayesian inference of epistatic interactions in case-control studies," *Nature genetics*, vol. 39, no. 9, pp. 1167–1173, 2007.

[62] A. Albert, *Regression and the Moore-Penrose pseudoinverse*. Elsevier, 1972.

[63] G. H. Golub and C. Reinsch, "Singular value decomposition and least squares solutions," *Numerische Mathematik*, vol. 14, no. 5, pp. 403–420, 1970.

[64] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[65] D. F. Schwarz, I. R. König, and A. Ziegler, "On safari to random jungle: a fast implementation of random forests for high-dimensional data," *Bioinformatics*, vol. 26, no. 14, pp. 1752–1758, 2010.

[66] S. J. Winham, C. L. Colby, R. R. Freimuth, X. Wang, M. de Andrade, M. Huebner, and J. M. Biernacka, "Snp interaction detection with random forests in high-dimensional genetic data," *BMC bioinformatics*, vol. 13, no. 1, p. 164, 2012.

[67] John, "How do i set the cuda stream cula uses?." `http://www.culatools.com/forums/viewtopic.php?f=14&t=1019`.

[68] "Magma documentation, function magmablassetkernelstream." `http://icl.cs.utk.edu/projectsfiles/magma/doxygen/group__magma__util.html#ga0bbe77c58765b97bf03f589c43cb7ac1`.

[69] PDC, "Zorn cluster." `https://www.pdc.kth.se/resources/computers/zorn`.

[70] "Plink data format." `http://pngu.mgh.harvard.edu/~purcell/plink/data.shtml`.

Appendices

Appendix A

Lists

List of Abbreviations

MAF
Minor allele frequency, the frequency of the least common allele.

RF
Random Forest

MDR
Random Forest

OR
Random Forest

RR
Random Forest

RERI
Random Forest

AP
Random Forest

SI
Random Forest

GPU
Random Forest

CPU
Random Forest

SM
Streaming multiprocessor

SIMD
Random Forest

MIMD
Random Forest

SIMT
Random Forest

SISD
Random Forest

LR
Random Forest

MIDSD
Random Forest

List of Algorithms

List of Examples

List of Figures

List of Tables

Appendix B

File Formats

B.1 PLINK Data Format

The PLINK files can be in different formats.
The binary format consists of three files. Bed, bim and fam. The
[70]

B.2 Environmental and Covariates File Format

The environmental factors and covariates are stored in separate files with the same format, one file for environmental factors and one for the covariates if any. One of the columns needs to contain the individual ids, the rest of the columns should be data. The delimiter can be any reasonable string(e.g. something that is not part of a name or number) and there is an option to set the delimiter in the program. The default is tab delimited.

B.3 CuEira Result File Format

The results from CuEira are written in an csv file. It has 48 columns described below.

snp_id
: The id of the SNP from the bim file.

pos
: The row which the SNP had in the plink files.

skip
: Numbers explaining why the SNP was excluded from calculation. 1=missing data. 2=low MAF, 3=low cell count, 4=negative position in bim file

risk_allele
: The risk allele, not changed based on recoding

minor
: The minor allele

major
: The major allele

env_id
: The id of the environment factor, it is the column form the environment file.

no_alleles_X
: The numbers of the alleles in group X

freq_alleles_X
: The frequencies of the alleles group X

no_snpX_envY
: Cell distribution with exposure X and Y.

ap
: The AP value for the additive model described in 2.1.4

reri
: The RERI value for the additive model described in 2.1.4

OR
: The odds ratios for the SNP, environment factor and interaction. Add is for the additive model and mult is for the multiplicative. L is the lower confidence interval and H is the upper. The interval is from using the delta method[12].

recode
: Case of recoding, 0=no recoding, 1=snp protective, 2=environment protective, 3=interaction protective

Each line after the header is a combination of a SNP and an environment factor. Because of the parallel nature of the program the rows are not in any specific order. The column pos contains the row number that the SNP had in the plink files. This can be used to sort the data in.

The data in some columns are changed depending on the recoding. The multiplicative model is calculated using the recode from the additive model. The cell frequencies are also changed. However the risk allele is not changed.

Appendix C

Risk Allele

This appendix contains a more detailed look at different risk allele definitions than section 1.3.1.
The code for how JEIRA and GEISA calculates the risk allele is not the same as its definition [3, 12] talked about previously in section 1.3.1. The algorithm JEIRA and GEISA uses is shown in algorithm 4.

Algorithm 10: JEIRA and GEISA pseudo code to determine the risk allele

Data: $caseMaxAllele$ is the most common allele in case and
$controlMaxAllele$ is the most common in control
$caseMaxRatio$ and $controlMaxRatio$ is the frequency of
$caseMaxAllele$ and $controlMaxAllele$ respectively.

if $caseMaxRatio > controlMaxRatio$ **and**
$caseMaxAllele = controlMaxAllele$ **then**
  $riskAllele \longleftarrow caseMaxAllele$
else
  $riskAllele \longleftarrow caseMinAllele$

The allele frequencies can be shown in an 2×2 table. The columns are the frequencies for the different alleles and the rows are the groups(case and control). It can be split into four cases shown in table C.1. The four cases comes from the four cases of possible directions of the inequality between the maximum frequencies of the two groups. There is up, down, down diagonally and up diagonally. The rest of the inequalities can be found by using that the sum of each row is 1.

By using the previously mentioned definitions in section 1.3.1 and the JEIRAs and GEISAs algorithm 4 what the risk allele is according to them for each case is shown in table C.2. Each definition correspond to examining the inequalities in the cases. For the definition using MAF the direction of the inequality from case minimum to control minimum determines it. If it points towards control the the case minor allele is the risk allele. For JEIRA and GEISA if the

	Allele1		Allele2		Allele1		Allele2
Case	Max	$\rightarrow$	Min		Max	$\rightarrow$	Min
	$\downarrow$	$\searrow\!\!\!\nearrow$	$\uparrow$		$\uparrow$	$\searrow\!\!\!\nearrow$	$\downarrow$
Control	Max	$\rightarrow$	Min		Max	$\rightarrow$	Min
Case	Max	$\rightarrow$	Min		Max	$\rightarrow$	Min
	$\downarrow$	$\searrow\!\!\!\nearrow$	$\uparrow$		$\downarrow$	$\swarrow\!\!\!\nwarrow$	$\uparrow$
Control	Min	$\leftarrow$	Max		Min	$\leftarrow$	Max

Table C.1: *The four cases of allele frequencies*

maximum for case points straight down to the maximum for control then the major case allele is the risk allele. This only happens in case 1. For increased frequency in case it is the allele which has the inequality pointing straight down.

Definition	Case 1	Case 2	Case 3	Case 4
MAF based	Allele1	Allele2	Allele1	Allele2
JEIRA GEISA algorithm	Allele1	Allele2	Allele2	Allele2
Increase frequency in case	Allele1	Allele2	Allele1	Allele1

Table C.2: *Risk alleles for the cases in table C.1 based on the different definitions*

The results are different for case 3 and case 4.
For case 4 it is clear that Allele1 should be the risk by looking at an extreme case. The case is shown in table C.3. Almost everyone in the case group has Allele1 while no one in the control group does. The MAF based definition and JERIA/GEISA sets Allele2 as the risk for this case while increase in frequency sets Allele1.

	Allele1	Allele2
Case	0.99	0.01
Control	0	1

Table C.3: *Example frequencies of case 4 in table C.1*

Appendix D

Bugs and Issues

This is a small collection of various bugs and limitations encountered in the course of the project that can be good to know for anyone that wants to compile CuEira or if the libraries are used in other projects.

D.1 Bugs

There are several problems with Intel compilers and libraries. Most libraries work fine with gnu C/C++ compiler but have problems with Intel compilers. Boost 1.55 does not work with Intel compilers, 1.54 works. AllOf in Google Test also does not work together with Intel compilers.

JEIRA prints the wrong results for the multiplicative model and as mentioned in appendix C the risk allele algorithm JEIRA uses is not the same as the definition.

D.2 Issues

Nvcc can not compile c+11 code and no interface included in the code that nvcc compiles can use c++11. This problem can largely be avoided by using separate compilation. The CUDA find package module for CMake(included in version 2.8 and later) has a default flag that should be removed as it can cause problems. The flag tells nvcc to propagate the host code flags, if the compiler for the host code then uses c++11 that flag is propagated to nvcc and it will not compile The flag is disabled in CMake by using set(CUDA_PROPAGATE_HOST_FLAGS off).

Appendix E

How to Compile and Use CuEira

The source code for the program is available at github, `https://github.com/Berjiz/CuEira`.

List of dependencies. Listed version is the version used.

- Boost, tested with version 1.54, 1.55 does not work due to a bug with Intel compilers
- CMake, version 2.8
- CUDA, version 5.5
- MKL, version 13.1
- Intel compiler version 14.0
- Google Test and Google Mock, already included in the folder.

First install the dependencies, they might need to be compiled from source. Open a terminal and make a new directory where you want the program to be compiled. Enter the directory and write:
cmake /path/to/CuEira/
Then type:
make

The program should now be compiling. CMake might complain about not finding a dependency. If it does you need to tell CMake where it is. The path to boost is set by using EXPORT BOOST_ROOT = /path/to/boost/. The path to the compilers is done with CXX = cmake /path/to/source for the C++ compiler and C = cmake /path/to/source for the C compiler.

In the build/bin directory there is two executable files, CuEira and CuEira_Test. CuEira is the program itself while CuEira_Test runs all the tests for the program.